CARL VON OSSIETZKY universität OLDENBURG

DEPARTMENT FÜR INFORMATIK
SYSTEMSOFTWARE UND VERTEILTE SYSTEME

Konzeption und Implementierung eines Rahmenwerkes für Replikationsstrategien in Sensornetzwerken

Diplomarbeit

26. November 2010

verfasst und vorgelegt von

Christoph Korinke

Dr.-Pickart-Str. 4

27793 Wildeshausen

zur Erlangung des akademischen Grades

Diplom-Informatiker

Erstprüfer: Prof. Dr.-Ing. Oliver Theel

Zweitprüfer: Dipl.-Ing. Jens Kamenik

Kurzfassung

Messaufgaben mit Sensornetzwerken, bei denen die Basisstation nicht ständig mit dem Sensornetzwerk verbunden ist (z.B. Vulkan- oder Gletscherbeobachtung), erfordern die Speicherung der Messwerte im Netzwerk bis zur Abholung der Daten. Da die Wahrscheinlichkeit für Fehler innerhalb von Sensornetzen größer als in klassischen drahtgebundenen Systemen ist, müssen die Messdaten im Sensornetzwerk repliziert werden. Eine Datenreplikation wird eingesetzt, um ein eventuelles Fehlschlagen von Zugriffen auf dringend benötigte Daten möglichst auszuschließen. Die notwendigen Ressourcen, um genügend Redundanz zu erzeugen, stehen in Sensornetzwerken aber nur begrenzt zur Verfügung. So ist der Energieverbrauch in Sensornetzen ein entscheidender Faktor für die Lebensdauer des gesamten Netzwerks.
Diese Diplomarbeit beschäftigt sich mit der Anpassung und Erweiterung eines generalisierten Verfahrens für Replikationsstrategien drahtgebundener Systeme an Sensornetzwerke. Das entstandene Konzept wird prototypisch implementiert und das Energiesparpotential im Vergleich mit klassischen Strategien im Simulator untersucht. Die Architektur der Implementierung ist dabei so gewählt, dass ein Austausch der Replikationsverfahren einfach möglich ist.

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.

Wildeshausen, den 26. November 2010

Christoph Korinke

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Kapitel 1

Einleitung

Drahtlose Sensornetzwerke werden zur Beobachtung von Ereignissen der realen Welt, wie zum Beispiel der Temperatur, eingesetzt. Es gibt viele verschiedene Anwendungsbereiche, wie die Brandentdeckung in Waldgebieten oder die Gletscherbeobachtung. Dabei übernehmen einzelne Sensorknoten nur einfache Aufgaben, wie das Auslesen und Versenden der Sensordaten. Erst das Zusammenspiel mehrerer Knoten in einem Sensornetz erlaubt es komplexere Aufgaben und Beobachtungen durchzuführen.

Häufig ist eine Basisstation mit dem Sensornetzwerk permanent verbunden, die die Daten entgegen nimmt und verarbeitet. Bei bestimmten Anwendungen, wie der Gletscher- oder Vulkanbeobachtung, ist die Basisstation jedoch nicht stationär und dadurch nicht ständig mit dem Sensornetz verbunden. Hierbei ist es erforderlich die Messwerte im Netzwerk bis zu ihrer Abholung zu speichern. Da die Wahrscheinlichkeit für Fehler innerhalb von Sensornetzen größer als in klassischen drahtgebundenen Systemen ist, müssen die Messdaten im Sensornetzwerk repliziert werden.

Dazu können Replikationsstrategien eingesetzt werden, die kritische Daten redundant und konsistenzerhaltend an mehreren Orten in einem Netzwerk speichern. Dabei sollen neben der hohen Verfügbarkeit der Daten, die Operationskosten gering bleiben. Die Operationskosten ergeben sich unter anderem aus der Anzahl benötigter Nachrichten. Ein großes Problem von Sensornetzen ist die Energieeffizienz, wobei die Kommunikation zwischen den Sensorknoten den Energieverbrauch überwiegt. Ursprünglich sind Replikationsstrategien jedoch für Datenbanken entwickelt und damit nicht auf die Anforderungen von Sensornetzwerken zugeschnitten.

Ziel dieser Arbeit ist es, Replikationsstrategien auf Sensornetze zu übertragen und so anzupassen, dass die Daten weiterhin konsistent im Netzwerk verteilt werden, dabei aber die Lebensdauer des gesamten Netzwerks im Verhältnis zu klassischen Strategien erhöht wird. Dafür wird ein prototypisches Rahmenwerk implementiert, mit dem sowohl die klassischen als auch die angepassten Verfahren getestet und deren Energieverbrauch verglichen und untersucht werden kann. Dabei ist die Architektur der Implementierung für einen leichten Strategieaustausch ausgelegt.

In Kapitel 2 sind die benötigten Grundlagen für die Arbeit dargelegt. Zuerst werden Sensornetze und ihre Sensorknoten und verschiedene Betriebssysteme genauer betrachtet. Danach sind Möglichkeiten zum Einsparen von Energie beschrieben. Nach den Grundlagen für die Sensornetze, wird auf Replikationsstrategien eingegangen. Dafür werden zuerst Vorgehensweisen bei der Kommunikation und wichtige Begriffe erläutert und danach verschiedene klassische

Verfahren vorgestellt. Das Kapitel schließt mit einer Verallgemeinerung der Verfahren und einer Auflockerung der Konsistenzbedingungen von Replikationsstrategien ab.

Nachdem alle nötigen Grundlagen erklärt sind, folgt in Kapitel 3 das zielführende Konzept, das ein generalisierendes Verfahren für klassische Replikationsstrategien an Sensornetzwerke anpasst. Dazu werden zunächst die Anforderungen an das Konzept definiert und diese dann schrittweise umgesetzt. Daneben wird das Verfahren zur Energieeinsparung um eine Struktur erweitert, das eine gezieltere Auswahl der Replikationsorte ermöglicht.

Das Konzept wird in einem Rahmenwerk umgesetzt, das in Kapitel 4 zu finden ist. Dabei ist zuerst ein Überblick über die Architektur gegeben, danach werden die einzelnen Teile des Rahmenwerkes genauer betrachtet. Das Kapitel schließt mit Designentscheidungen, Erweiterungs- und Optimierungsmöglichkeiten und Funktionstests ab.

Die Simulation des Rahmenwerkes zur Überprüfung des Konzeptes ist in Kapitel 5 beschrieben. Dafür wird zuerst der Simulator vorgestellt und dieser mit einem Testmodell kombiniert. Darauf folgt die Beschreibung der Testszenarien, für die vor der Simulation Annahmen zum Energieverbrauch getroffen werden. Zum Schluss des Kapitels werden die Simulationsergebnisse vorgestellt und mit den Annahmen verglichen.

Die Arbeit schließt im letzten Kapitel mit einer Zusammenfassung der Ergebnisse ab und gibt einen kurzen Ausblick auf zukünftige Arbeiten und Verbesserungen.

Kapitel 2

Grundlagen

Dieses Kapitel bietet eine Einführung in Sensornetze und Replikationsstrategien und stellt die Grundlage dar, auf die diese Arbeit aufgebaut ist. Dafür werden zuerst Sensornetze und ihre Sensorknoten betrachtet. Da auf den Sensorknoten in der Regel ein Betriebssystem läuft, werden drei Betriebssysteme für Sensorknoten vorgestellt, wobei auf das in dieser Arbeit benutzte *TinyOS* intensiver eingegangen wird. Vor der Beschreibung von Replikationsstrategien werden noch Möglichkeiten zur Energieeinsparung bei Sensornetzen behandelt.

Nach den Sensornetzen folgt eine Einführung in Replikationsstrategien, die zur konsistenten Verteilung von Daten dienen. Dafür wird zuerst auf Kommunikationsprotokolle eingegangen und darauf Bedingungen für eine konsistente Verteilung definiert. Vor der Beschreibung der klassischen Replikationsstrategien, werden noch die Begriffe Verfügbarkeit, Kosten und Partitionierung kurz erläutert. Nach den klassischen Strategien werden zwei erweiterte Replikationsverfahren vorgestellt. Das erste Verfahren generalisiert dabei den Ansatz der klassischen Strategien. Der zweite Ansatz arbeitet mit Wahrscheinlichkeiten, um eine erhöhte Verfügbarkeit der Daten zu erreichen, wobei jedoch die vollständige Konsistenz der Daten aufgegeben wird.

2.1 Drahtlose Sensornetze

Drahtlose Sensornetze dienen der weiträumigen Beobachtung von Ereignissen und Phänomenen der realen Welt, wie zum Beispiel Temperatur, Beschleunigung oder Luftfeuchtigkeit. Sie werden in vielen Anwendungsbereichen eingesetzt, wie der Brandentdeckung in Waldgebieten oder zur Vulkan- und Gletscherbeobachtung [SMZ07]. Während ein einzelner Sensorknoten nur einfache Aufgaben, wie das Auslesen und Weiterleiten von Sensordaten übernimmt, lassen sich durch Zusammenwirken mehrerer Knoten in einem Sensornetz komplexe Aufgaben und Beobachtungen durchführen.

Ein großes Problem von Sensornetzen ist die Energieeffizienz [MR03]. Die Kommunikation mit anderen Knoten überwiegt dabei den Energieverbrauch der einzelnen Knoten. Verfahren zur Energieeinsparung sollten nicht nur den Verbrauch einzelner Knoten minimieren, sondern die Energiebelastung so auf das Sensornetz verteilen, dass eine lange Lebenszeit garantiert wird. In Abschnitt 2.1.3 wird kurz auf die Energiekosten und Reduzierungsmöglichkeiten eingegangen. Neben der Energieeffizienz sind Verlässlichkeit und Fehlertoleranz wichtig [MR03], um in lebenswichtigen Anwendungen auch bei einem Ausfall einzelner Komponenten oder dem Auftreten von zu erwartenden Fehlern bestimmte Funktionsgarantien zu erfüllen.

Nachfolgend sind einige Unterschiede zwischen Sensornetzen und klassischen verteilten Systemen dargestellt [KW05][MR03]:

Energie Der größte Unterschied von Sensornetzen zu klassischen verteilten Systemen ist die Energie. Während klassische Systeme überwiegend an Steckdosen angeschlossen sind, besitzen Sensornetze autarke Energiequellen, wie beispielsweise Batterien. Aufgrund der Menge der Sensorknoten und deren Standorte lassen sich die Batterien meist nicht ersetzen. Somit sind Sensornetze auf energieeffiziente Algorithmen angewiesen, um eine lange Lebensdauer zu garantieren.

Anwendungen & Ausstattung Sensorknoten bestehen aus wenigen Komponenten und sind nur für bestimmte Aufgaben ausgelegt. Klassische verteilte Systeme hingegen bestehen aus Servern, Laptops oder Desktop-PCs und bieten viele Anwendungsmöglichkeiten mit demselben System.

Mobilität Durch Umwelteinflüsse können die Standorte von Sensorknoten beeinflusst werden. Dadurch muss das Sensornetz Kommunikationspfade selbstständig anpassen können.

Kommunikationsprotokolle Messdaten von nahe gelegenen Sensorknoten sind oft korreliert, wodurch Fehler entdeckt werden können. Kommunikationsprotokolle, wie TCP/IP, die eine korrekte Übertragung garantieren, sind deswegen oftmals nicht notwendig. Dadurch wird der Kommunikationsaufwand geringer und weniger Energie wird verbraucht.

Skalierbarkeit Sensornetze bestehen aus weit mehr Knoten als klassische verteilte Systeme. Sie können aus einigen Hunderten bis einigen Tausenden bestehen.

Eine direkte Interaktion der einzelnen Sensorknoten mit dem Benutzer ist in der Regel in einem Sensornetzwerk nicht vorgesehen, da diese normalerweise autark arbeiten. Stattdessen wird eine sogenannte Basisstation als Schnittstelle genutzt, über die ein Benutzer mit dem Netzwerk interagieren kann, um beispielsweise Sensordaten abzufragen.

Bei Sensornetzen, die nicht permanent mit einer Basisstation verbunden sind, müssen die einzelnen Knoten ihre Werte bis zur Abholung speichern. Bei einem Ausfall eines Knotens, zum Beispiel durch Umwelteinflüsse, wären die gespeicherten Daten verloren. Diese Diplomarbeit versucht unter Verwendung von Replikationsstrategien dem entgegen zu wirken.

Im nächsten Abschnitten wird genauer auf die Sensorknoten eingegangen und in Abschnitt 2.1.2 werden einige Betriebssysteme für Sensorknoten vorgestellt. Der letzte Abschnitt beschäftigt sich mit Energiekosten und wie diese niedrig gehalten werden können.

2.1.1 Sensorknoten

Sensornetze bilden eine Struktur aus einer Menge von Sensorknoten, die gewöhnlich aus einer oder mehreren Sensoreinheiten, einem Mikrocontroller, einer drahtlosen Kommunikationseinheit und einer autarken Energiequelle, wie Batterien, bestehen [SMZ07]. Der Mikrocontroller läuft in der Regel mit einer Taktrate von 15 bis $25MHz$ und besitzt Speichervolumen zwischen 32 und $128kB$. Zusätzlich zu den genannten Komponenten können je nach Anwendungsgebiet noch weitere Komponenten, wie Aktuatoren und Positionsbestimmungseinheiten, enthalten sein. Siehe hierzu auch Abbildung 2.1. Die Abbildung 2.2 zeigt einen realen Sensorknoten.

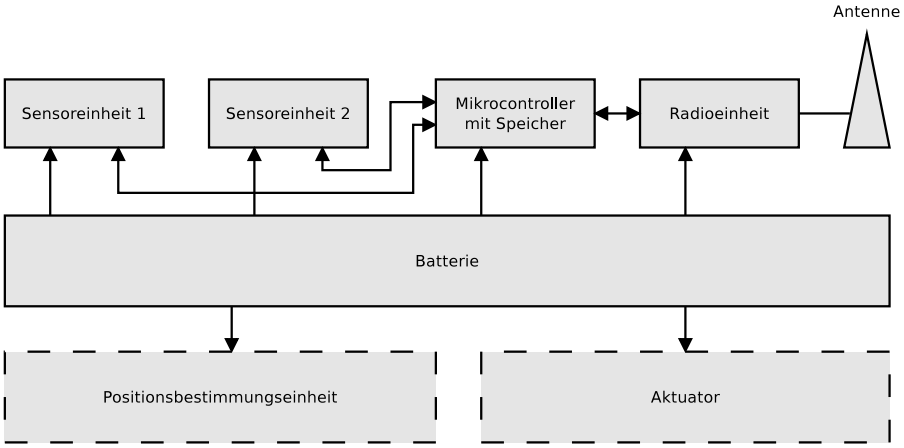


Abbildung 2.1: Typischer Aufbau eines Sensorknotens [SMZ07]

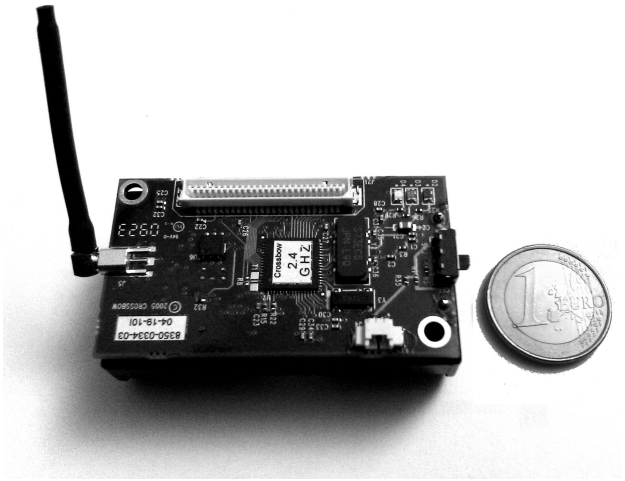


Abbildung 2.2: Sensorknoten vom Typ MicaZ

Sensorknoten, die eine Schnittstelle zwischen Netzwerk und Benutzer anbieten, werden als Basisstation bezeichnet. Die Basisstation ist dabei zum Beispiel direkt an einen Rechner angeschlossen oder mit dem Internet verbunden. Die Basisstation ermöglicht dem Benutzer auf Sensordaten zuzugreifen oder Konfigurationen im Netzwerk zu verteilen.

Bei der Kommunikation in einem Sensornetz kann zwischen einer Multi-Hop Kommunikation und einer Single-Hop Kommunikation unterschieden werden. Bei der Single-Hop Kommunikation wird eine Nachricht direkt vom Sender zum Empfänger übertragen. Dafür muss der Empfänger im Sendebereich des Senders liegen. Die Funkeinheit der Sensorknoten besitzt jedoch nur eine gewisse Reichweite, so dass eventuell nicht alle Knoten direkt miteinander kommunizieren können. Dieses Problem wird umgangen, indem Nachrichten über andere Knoten bis zum Empfänger weitergeleitet werden. In diesem Fall spricht man von einer Multi-Hop Kommunikation. Abbildung 2.3 auf der nächsten Seite zeigt einen typischen Aufbau eines Sensornetzwerks.

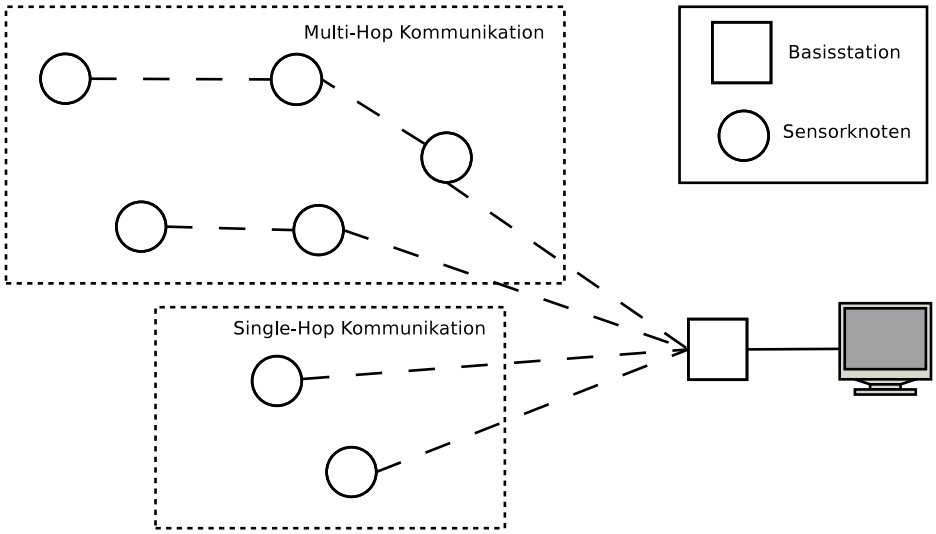


Abbildung 2.3: Aufbau eines Sensornetzes mit Multi-Hop und Single-Hop Kommunikation

2.1.2 Betriebssysteme

Auf jedem Sensorknoten läuft ein Betriebssystem, das die Hardware steuert und als Vermittler zwischen der Hardware und den eigentlichen Anwendungen agiert. Da Sensorknoten im Gegensatz zu PCs nur wenige Ressourcen, langsame Prozessoren und wenig Speicher haben, gelten andere Anforderungen an Betriebssysteme für Sensorknoten als bei traditionellen Betriebssystemen.

Die Anforderungen an Betriebssysteme für Sensorknoten werden kurz in [SMZ07] auf Seite 274 ff. erläutert und bestehen aus den folgenden Punkten:

- Da nur wenig Speicher zur Verfügung steht, sollte das Betriebssystem kompakt und klein sein.
- Es sollte Echtzeit-Fähigkeiten besitzen, da Informationen möglicherweise schnell veraltet sind.
- Damit die CPU-Zeit und der limitierte Speicher effizient und fair genutzt werden, ist eine effiziente Ressourcen-Verwaltung nötig.
- Eine zuverlässige und effiziente Code-Verteilung sollte angeboten werden, da sich die Anwendung eines Sensorknotens nach dem Auslegen am Anwendungsort ändern könnte.
- Um die durch die Batterien begrenzte Lebenszeit eines Knotens zu verlängern, ist eine Energieverwaltung sinnvoll, die den Knoten in einen Schlafzustand versetzt, wenn dieser keine Arbeit zu erledigen hat.
- Es sollte eine generische Programmierschnittstelle bieten, um einen direkten Zugriff auf die Hardware zu bekommen und so eine Optimierung des Systems zu ermöglichen.

Nachfolgend werden drei Betriebssysteme aus [SMZ07] vorgestellt, wobei das Letzte in dieser Arbeit eingesetzt wird und deswegen genauer beschrieben wird.

EMERALDS

EMERALDS ist ein Betriebssystem für verteilte eingebettete Echtzeit-Systeme, die auf langsamen Prozessoren mit wenig Speicherressourcen laufen. Es basiert auf einem erweiterbaren Mikrokernel, der in C++ geschrieben wurde. Ein Mikrokernel bietet nur die grundlegenden Funktionen der Speicher- und Prozessverwaltung, sowie der Synchronisation und Kommunikation. Dem entgegen steht der sogenannte monolithische Kernel, der zusätzlich Treiber für Hardwarekomponenten und weitere Funktionsbibliotheken implementiert.

EMERALDS bietet Multithreading und Speicherschutz für Prozesse. Die Prozesse werden nach den Echtzeit-Verfahren *Deadline First* und *Rate Monotonic* ausgeführt. Bei *Deadline First* wird der Prozess bevorzugt, dessen vorgegebener spätester Beendigungszeitpunkt als nächstes auftritt. Bei *Rate Monotonic* erhalten die Prozesse eine Priorität nach ihrer Periodendauer. Die Periodendauer gibt den Zeitpunkt einer erneuten Ausführung des Prozesses an. Je kürzer die Periodendauer, desto höher ist die Priorität des Prozesses und Prozesse mit einer höheren Priorität werden bevorzugt behandelt.

SenOS

Betriebssysteme die als Zustandsmaschinen, auch endliche Automaten genannt, realisiert sind, eignen sich besonders für parallele Ausführung, Synchronisation und Reaktionsfähigkeiten von Prozessen[SMZ07]. Ein endlicher Automat besteht aus Zuständen und Zustandsübergängen. Tritt ein Ereignis auf und die Bedingungen für einen Zustandsübergang sind gegeben, wechselt der Automat von dem aktuellen Zustand in den nächsten Zustand. Ein Automat wird als endlich bezeichnet, wenn er eine endliche Zustandsmenge besitzt, also nur endlich viele Zustände einnehmen kann.

SenOS basiert auf einem solchen endlichen Automaten und besteht aus den folgenden drei Komponenten:

Kernel Der Kernel besteht aus einer Zustandssteuerung und einer Ereignis-Warteschlange. Die Steuerung reagiert dabei auf die Warteschlange und arbeitet sie der Reihe nach ab.

Zustandsübergangstabellen Zustandsübergangstabellen legen fest, von welchem Zustand aus in welchen anderen Zustand gewechselt werden kann, welche Bedingungen dabei erfüllt sein müssen und welche Funktionen der Übergang aufruft. Jede Tabelle stellt eine eigene Anwendung dar.

Funktionsbibliothek Die Funktionsbibliothek enthält die Verweise auf die Funktionen der Anwendungen.

Der Kernel und die Funktionsbibliothek werden statisch gebaut und im Flashspeicher des Sensorknotens untergebracht. Die Zustandsübergangstabellen können hingegen zur Laufzeit modifiziert und neu geladen werden, um das Verhalten nach dem Auslegen der Knoten an neue Bedingungen anzupassen.

TinyOS

TinyOS ist ein ereignisgetriebenes Betriebssystem und besteht aus einzelnen Komponenten und einer Prozesssteuerung, die die Ausführung der Operationen der Komponenten verwaltet. Jede Komponente besteht wiederum aus einer Kommandosteuerung, einer Ereignissteuerung, einem Kontextrahmen und einer Menge von Aufgaben (Tasks). Außerdem deklariert jede Komponente ihre eigenen Kommandos und Ereignisse, die die Schnittstellen zur Interaktion mit der Komponente bilden.

Die *Prozesssteuerung*, im Folgenden Task Scheduler genannt, führt die Aufgaben nach dem First In First Out Prinzip aus. Zur Energieeffizienz veranlasst er den Prozessor in einen Schlafzustand zu wechseln, wenn keine Tasks mehr in der Warteschlange vorhanden sind und die Peripheriegeräte noch arbeiten. Der *Kontextrahmen* einer Komponente spezifiziert dessen Speicheranforderungen und wird während des Kompiliervorgangs mit einer festen Größe erzeugt. Die *Kommandos* sind nicht-blockierende Funktionsaufrufe, wodurch sie schnell ausgeführt werden. Sie bieten meistens einen Rückgabewert an, mit dem der Erfolg des Aufrufs ermittelt werden kann. Die ihm übergebenen Parameter speichert ein Kommando häufig im Kontextrahmen und reiht daraufhin einen Task in die Warteschlange des Tasks Schedulers ein. Auftretende *Ereignisse* werden durch die Ereignissteuerung, Event Handler genannt, verwaltet. Ein Event Handler kann Tasks einreihen, weitere Ereignisse (Events) auslösen und Kommandos aufrufen, um auf das Ereignis zu reagieren. *Tasks* können wie Events Kommandos aufrufen, Events auslösen und andere Tasks einreihen. Durch die bevorzugte Behandlung von Kommandos und Events werden Tasks bei deren Auftreten von ihrer Ausführung verdrängt.

Die *Komponenten* können in drei Typen eingeteilt werden. Der erste Typ abstrahiert die Hardware und bildet die unterste Ebene der Komponenten. Diese stellen somit den direkten Zugriff auf die Hardware dar. Der zweite Typ ist die künstliche Hardwarekomponente, die meistens auf einer Komponente des ersten Typs aufbaut und den Zugriff auf diese Komponente und somit auf die Hardware weiter vereinfacht. Die Komponenten, die die eigentlichen Anwendungen implementieren, werden als Softwarekomponenten bezeichnet und bilden den dritten Typ. Die Komponentenstruktur erlaubt einen leichten und schnellen Austausch von Funktionen und macht TinyOS modular und flexibel [SMZ07].

Da TinyOS für das in dieser Arbeit entwickelte Rahmenwerk eingesetzt wird, folgt nun eine kleine Einführung in die Implementierung und Verbindung von TinyOS Komponenten. Für eine genauere Darstellung sei hier auf [LG09] verwiesen. Die Komponenten für TinyOS sind in der Programmiersprache nesC geschrieben. nesC ist ein C Dialekt und stellt Konstrukte zur leichten Implementierung der Komponenten bereit. Des Weiteren werden die Komponenten in Module und Konfigurationen aufgeteilt. Ein Modul beinhaltet dabei die auszuführende Anwendung, während eine Konfiguration die Verbindungen der Komponenten über ihre Schnittstellen beschreibt.

Quelltext 2.1 zeigt beispielsweise das Modul *LedBootC*, das durch das Schlüsselwort `module` eingeleitet wird. Im `module`-Block werden die Schnittstellen definiert. Dabei gibt `uses interface` die Schnittstellen an, die die Komponente benutzt und `provides interface`, welche Schnittstellen die Komponente anbietet. *LedBootC* benutzt in diesem Fall die Schnittstellen *Boot* und *Led*.

Auf den `module`-Block folgt die Implementierung der Programmlogik, die durch das Schlüsselwort `implementation` eingeleitet wird. Innerhalb dieses Blocks können Funktionen in der Programmiersprache C geschrieben werden. Des Weiteren müssen an dieser Stelle alle Ereignisse (Events) der benutzten Schnittstellen implementiert werden. Im Quelltext 2.2 auf der nächsten Seite ist die Schnittstelle *Boot* abgebildet, die das Ereignis *booted* zur Verfügung stellt und das ausgelöst wird, sobald TinyOS gestartet ist. Das Modul *LedBootC* muss das Ereignis, wie in Zeile 8 zu sehen, implementieren. Tritt das Ereignis *booted* auf, wird in diesem Fall das Kommando *led0On* der *Led*-Schnittstelle durch den `call`-Befehl aufgerufen, wodurch die erste Led des Sensorknotens angeschaltet wird. Im Quelltext 2.2 ist neben der *Boot* Schnittstelle auch ein Ausschnitt der *Led* Schnittstelle dargestellt.

Die verwendeten Schnittstellen des *LedBootC* Moduls müssen im nächsten Schritt mit Komponenten verbunden werden, die diese Schnittstellen bereitstellen. Dazu dient die Konfiguration im Quelltext 2.3 auf der nächsten Seite mit dem Namen *LedBootAppC*.

Eine Konfiguration wird mit dem Schlüsselwort `configuration` gefolgt vom Namen der Konfiguration eingeleitet. Innerhalb dieses Blocks können wieder Schnittstellen definiert werden, die von der Konfiguration benutzt oder angeboten werden, indem sie diese an die Module weiterleitet. In dem Fall von *LedBootAppC* sind keine Schnittstellen angegeben. Nach dem `configuration`-Block folgt der `implementation`-Block. In diesem werden durch das Schlüsselwort `components` die zu verwendenden Komponenten definiert. *MainC* und *LedsC* sind mitgelieferte Komponenten von TinyOS. Dabei bietet *MainC* unter anderem die Schnittstelle *Boot* an und *LedsC* die Schnittstelle *Leds*. *LedBootC* ist die oben erstellte Komponente. Mittels `->` werden die Schnittstellen der Komponenten nun miteinander verbunden. Dabei zeigt der Pfeil immer vom Benutzer der Schnittstelle zum Anbieter der Schnittstelle.

```
1  module LedBootC
2  {
3    uses interface Boot;
4    uses interface Led;
5  }
6  implementation
7  {
8    event void Boot.booted()
9    {
10     call Leds.led0On();
11   }
12 }
```

Quelltext 2.1: nesC Modul LedBootC

```
interface Boot
{
  event void booted();
}

interface Leds
{
  command void led0On();
  command void led0Off();
  command void led0Toggle();
  ...
}
```

Quelltext 2.2: nesC Schnittstellen von Boot und Led

```
configuration LedBootAppC
{
}
implementation
{
  components MainC, LedsC, LedBootC;

  LedBootC.Boot -> MainC.Boot;
  LedBootC.Leds -> LedsC.Leds;
}
```

Quelltext 2.3: nesC Konfiguration LedBootAppC

Nach dieser kurzen Einführung in TinyOS werden nun die Energiekosten genauer betrachtet, die in Sensornetzen eine große Rolle spielen.

2.1.3 Energiekosten

Wie vorhergehend beschrieben werden Sensorknoten in der Regel mit Batterien betrieben. Diese besitzen jedoch nur eine geringe Kapazität, weswegen der Energiekonsum der Knoten kontrolliert werden sollte. Der Prozessor durch die Datenverarbeitung, die Radio-Einheit durch die Kommunikation, der Zugriff auf Speicher und das Auslesen der Sensoren sind dabei die Hauptverbraucher [KW05][SMZ07].

Um Energie einzusparen werden deswegen Niedrig-Energie-Chips (low power chips) in den Sensorknoten eingesetzt. Diese bieten verschiedene Zustände mit verringertem Energieverbrauch an. Als Gegenleistung ist jedoch die Funktionalität in diesen Zuständen eingeschränkt. Da die Sensorknoten die meiste Zeit bis zum nächsten Auslesen der Sensoren keine Aufgaben verrichten müssen, ist die Grundidee die Komponenten währenddessen in eine Art Schlafzustand zu versetzen, um Energie zu sparen. Erst wenn Ereignisse auftreten, sollen sie wieder ihren vollen Funktionsumfang erreichen.

Je nach Chip gibt es verschiedene Arten der Zustände. Bei Mikrocontrollern sind die typischen Zustände *Aktiv (active)*, *Warten (idle)* und *Schlafen (sleep)*. Je *tiefer* ein Schlafzustand ist, desto weniger Energie wird verbraucht. Der Vorteil dieser Chips geht jedoch verloren, wenn sie nicht sachgemäß eingesetzt werden, wie nachfolgend auf Basis von [KW05] erläutert wird.

Jeder Übergang von einem Zustand zu einem anderen verbraucht Energie und Zeit. Je tiefer der Schlafzustand ist, desto mehr Zeit und Energie wird benötigt, um wieder in einen weniger tiefen Schlafzustand oder den aktiven Zustand zu gelangen, in dem alle Operationen in vollem Umfang verfügbar sind. Gehen wir nun von einem Mikrocontroller mit den drei eben genannten Zuständen aus, kann es sich eventuell aus energetischer Sicht lohnen im idle-Zustand zu verharren, anstatt in den sleep-Zustand zu wechseln. Dies kann anhand von Abbildung 2.4 sichtbar gemacht werden, das ein allgemeines Energiemodell darstellt und [KW05] entnommen ist.

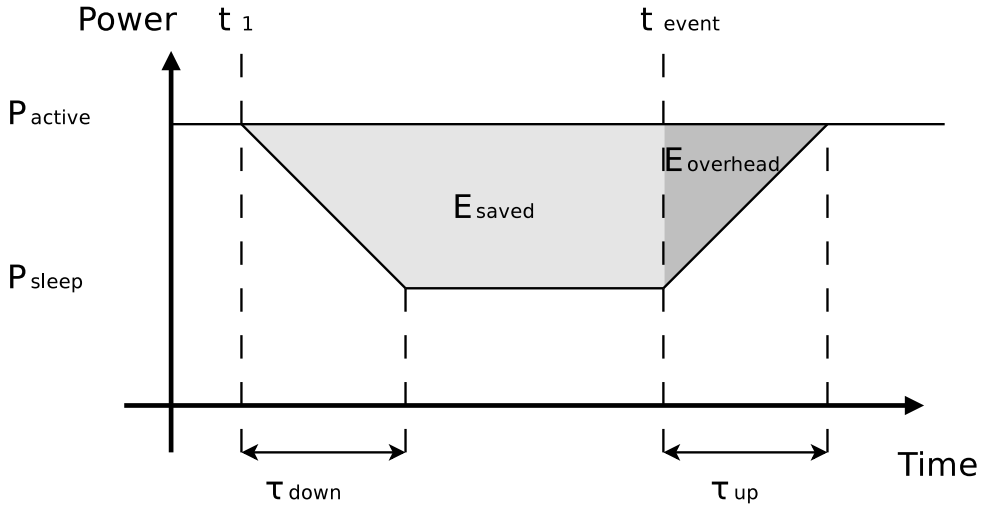


Abbildung 2.4: Energieeinsparung im sleep-Zustand [KW05]

Zum Zeitpunkt t_1 soll entschieden werden, ob eine Komponente in den Schlafzustand versetzt wird oder nicht, um damit den Energieverbrauch von P_{active} auf P_{sleep} zu reduzieren. Falls sich gegen einen Zustandswechsel entschieden wird und zum Zeitpunkt t_{event} ein Ereignis auftritt, das bearbeitet werden muss, wurde $E_{active} = (t_{event} - t_1) * P_{active}$ Energie verbraucht.

Wird die Komponente zum Zeitpunkt t_1 in den sleep-Zustand versetzt, wird eine Zeit τ_{down} benötigt, um diesen Zustand zu erreichen. Zur Vereinfachung wird ein mittlerer Energieverbrauch von $(P_{active} + P_{sleep})/2$ während dieser Phase angenommen. Bis zum Zeitpunkt t_{event} wird nun nur noch P_{sleep} konsumiert. Im sleep-Zustand ergibt sich damit ein Energieverbrauch zwischen t_1 und t_{event} von $E_{sleep} = \tau_{down} * (P_{active} + P_{sleep})/2 + (t_{event} - t_1 - \tau_{down}) * P_{sleep}$.

Die gesparte Energie in diesem Bereich ist also:

$$\begin{aligned} E_{saved} =& E_{active} - E_{sleep} \\ =& (t_{event} - t_1) * P_{active} - \left(\tau_{down} * \frac{(P_{active} + P_{sleep})}{2} + (t_{event} - t_1 - \tau_{down}) * P_{sleep}\right) \end{aligned}$$

Da die Komponente beim Auftreten des Ereignisses zum Zeitpunkt t_{event} aus dem sleep-Zustand wieder zurück in den active-Zustand versetzt werden muss, tritt ein Overhead in Form von zusätzlichen Energiekosten auf. Bei dem Zustandswechsel τ_{up} wird wie bei τ_{down} eine Vereinfachung des Energieverbrauchs angenommen. Somit ist ein Wechsel in den sleep-Zustand nur nützlich wenn $E_{overhead} < E_{saved}$.

Eine weitere Möglichkeit um Energie zu sparen besteht darin, die Chips mit einer geringeren Taktrate zu betreiben, die ausreicht, um die ihr zugewiesenen Aufgaben noch in einer geforderten Zeit zu erledigen. Eine geringere Taktrate verursacht einen geringeren Energieverbrauch [KW05]. Des Weiteren verursacht eine Multi-Hop Kommunikation im Allgemeinen weniger Energiekosten, als eine Single-Hop Kommunikation, da bei einer direkten Übertragung mehr Energie zur Vergrößerung des Senderadius nötig wird [SMZ07].

In den nächsten Abschnitten werden nun die Grundlagen von Replikationsstrategien vorgestellt.

2.2 Replikationsstrategien

Replikationsstrategien speichern kritische Daten redundant und konsistenzerhaltend an mehreren Orten. Dadurch wird in verteilten Systemen die Verfügbarkeit der Daten bei Auftreten von Fehlern erhöht [Her86]. Fehler sind ausgefallene Rechner oder Unterbrechungen innerhalb der Kommunikationswege. Ursprünglich wurden Replikationsstrategien für Datenbanken entwickelt. In dieser Ausarbeitung werden sie jedoch auf Sensornetze übertragen. Die Replikation sollte transparent sein, d.h. der einzige zu beobachtende Effekt ist die erhöhte Verfügbarkeit. Auch der Durchsatz kann durch verteilte Operationen auf den verteilten Daten erhöht werden [The02]. Die Kopien der Daten werden Replikate genannt. Die einzelnen Rechner im Netzwerk heißen Knoten.

Nach [Sto06] stellt ein Replikationsverfahren die gegenseitige Konsistenz der Replikate nach einem vorgegebenen Korrektheitskriterium sicher. Dabei sollen neben der hohen Verfügbarkeit der Daten, die Operationskosten, zum Beispiel Energieverbrauch, gering bleiben und die Lastverteilung der Zugriffe auf mehrere Rechner stattfinden. Die Operationskosten ergeben sich aus der Anzahl benötigter Nachrichten und den verwendeten Netzwerkverbindungen, die unterschiedliche Kosten verursachen.

Für verschiedene Anwendungsszenarien wurden mit der Zeit viele unterschiedliche Replikationsverfahren entwickelt. Jede Strategie hat Vor- und Nachteile und keine ist für alle Anwendungen gleichermaßen gut geeignet [Sto06]. Die Verfahren implementieren einen Kompromiss der Zielkonflikte, die in Abbildung 2.5 dargestellt sind. Je nach Anwendungsszenario wird auf einen Aspekt mehr oder weniger Wert gelegt.

Im nächsten Abschnitt werden Transaktionsprotokolle zur Kommunikation vorgestellt. Darauf folgt die Definition von Quoren, die eine Menge von Sensorknoten darstellen und für die konsistente Replizierung benutzt werden. Bevor in Abschnitt 2.2.5 klassische Replikationsverfahren beschrieben werden, wird noch in Abschnitt 2.2.3 auf den Verfügbarkeitsbegriff und in Abschnitt 2.2.4 auf Partitionierung eingegangen. Ein neuer Ansatz zu den klassischen Verfahren wird in Abschnitt 2.2.6 erläutert, bevor dann zuletzt probabilistische Verfahren erklärt werden,

Abbildung 2.5: Zielkonflikt von Replikationsstrategien [The02]

die die in Abschnitt 2.2.2 eingeführten Definitionen lockern. Die nachfolgenden Beschreibungen basieren auf [Sto06], [Dad96] und [The02] und können dort detaillierter nachgelesen werden.

2.2.1 Transaktionen

Transaktionsprotokolle dienen der Konsistenzerhaltung der Replikate. Alle Knoten, die an einer Schreiboperation beteiligt sind, müssen nach deren erfolgreichen Abschluss Replikate mit gleichem Wert besitzen. Eine Transaktion ist somit eine Folge von logisch zusammengehörenden Operationen, die ein Datum von einem konsistenten Zustand in einen konsistenten Folgezustand überführt. Dabei muss das ACID-Prinzip eingehalten werden, das aus folgenden vier Bedingungen besteht [Dad96]:

Atomarität (atomicity) Die zu einer Transaktion gehörende Operationsfolge ist eine logische Einheit und darf nur vollständig ausgeführt werden. Kann eine begonnene Transaktion nicht beendet werden, müssen alle bereits getätigten Änderungen rückgängig gemacht werden.

Konsistenz (consistency) Eine vollständig ausgeführte Transaktion darf die Konsistenz der Daten nicht verletzen.

Isolation (isolation) Zwei parallel ausgeführte Transaktionen auf dem selben Knoten dürfen sich nicht überschneiden. Wird eine Transaktion zurückgesetzt, ist keine andere Transaktion davon betroffen. Transaktionen auf einem Datum müssen somit serialisierbar sein.

Dauerhaftigkeit (durability) Die Änderungen einer erfolgreich abgeschlossenen Transaktion bleiben dauerhaft erhalten. Änderungen können nur durch weitere Transaktionen hervorgerufen werden.

Zur Einhaltung des ACID-Prinzips ist die Synchronisation der an der Transaktion teilnehmenden Knoten durch Commit-Protokolle nötig. Ein Knoten, der die Transaktion kontrolliert, ist der *Koordinator*. Alle ausführenden Knoten werden als *Teilnehmer* bezeichnet. Der Koordinator kann auch gleichzeitig ein Teilnehmer sein. Nachfolgend werden zwei Commit-Protokolle vorgestellt. Das Drei-Phasen-Commit-Protokoll ist dabei eine Erweiterung des Zwei-Phasen-Commit-Protokolls.

Zwei-Phasen-Commit-Protokoll

Bei dem Zwei-Phasen-Commit-Protokoll (2PC-Protokoll) stimmen alle an der Transaktion beteiligten Knoten ab, ob die Transaktion global abgeschlossen (*commited*) oder abgebrochen (*aborted*) wird. Es besteht aus der Wahlphase und der Entscheidungsphase [Dad96]:

Phase 1: Wahlphase Die Teilnehmer werden vom Koordinator aufgefordert den Commit der Transaktion vorzubereiten und ihr Abstimmungsergebnis zurück zu senden.

Phase 2: Entscheidungsphase Falls alle Knoten dem *Commit* zustimmen (*Commit-Fall*), meldet der Koordinator ein Commit an alle beteiligten Knoten. Falls mindestens ein Knoten die Zustimmung verweigert, sendet der Koordinator ein *Abort* an die beteiligten Knoten. Teilnehmer, die ihre Zustimmung verweigern, wechseln von alleine in den *Abort* Zustand.

Siehe hierzu auch Abbildung 2.6, die aus [Dad96] stammt.

Falls in der Wahlphase Teilnehmer ausfallen, müssen die verbleibenden Teilnehmer warten, da nicht bestimmt werden kann, wie die ausgefallenen Teilnehmer entschieden haben. Fällt der Koordinator in der Entscheidungsphase aus, sind alle Teilnehmer blockiert bis der Koordinator repariert ist und die Entscheidung bekannt gibt. Das 2PC-Protokoll ist somit ein blockierendes Protokoll. Eine genaue Beschreibung des Zwei-Phasen-Commit-Protokolls ist in [Dad96] auf den Seiten 206 bis 210 zu finden.

Diese Schwachstelle des 2PC-Protokolls hat unter anderem zum Drei-Phasen-Protokoll geführt, das nachfolgend beschrieben wird.

Drei-Phasen-Commit-Protokoll

Das Drei-Phasen-Commit-Protokoll (3PC-Protokoll) [Rah94] ist eine Weiterentwicklung des 2PC-Protokolls, um die Blockierung durch den Koordinator zu umgehen. Dafür wird nach der Entscheidungsphase eine neue Phase eingeführt. Bis zu dieser Phase ist das 3PC-Protokoll mit dem 2PC-Protokoll identisch.

Nach dem Empfang der positiven Abstimmungsergebnisse (*commit*) von allen Teilnehmern, sendet der Koordinator eine Nachricht zum Vorbereiten des Commits (*prepare to commit*) an die Teilnehmer. Diese wechseln in den *PreCommit* Zustand und antworten, dass sie bereit sind zum Abschließen der Transaktion. Sobald der Koordinator alle Antworten erhalten hat, sendet er eine Nachricht an alle Teilnehmer die Transaktion erfolgreich zu beenden. Die Teilnehmer wechseln danach in den Endzustand, heben alle gesetzten Sperren auf und sind für die nächste Transaktion verfügbar.

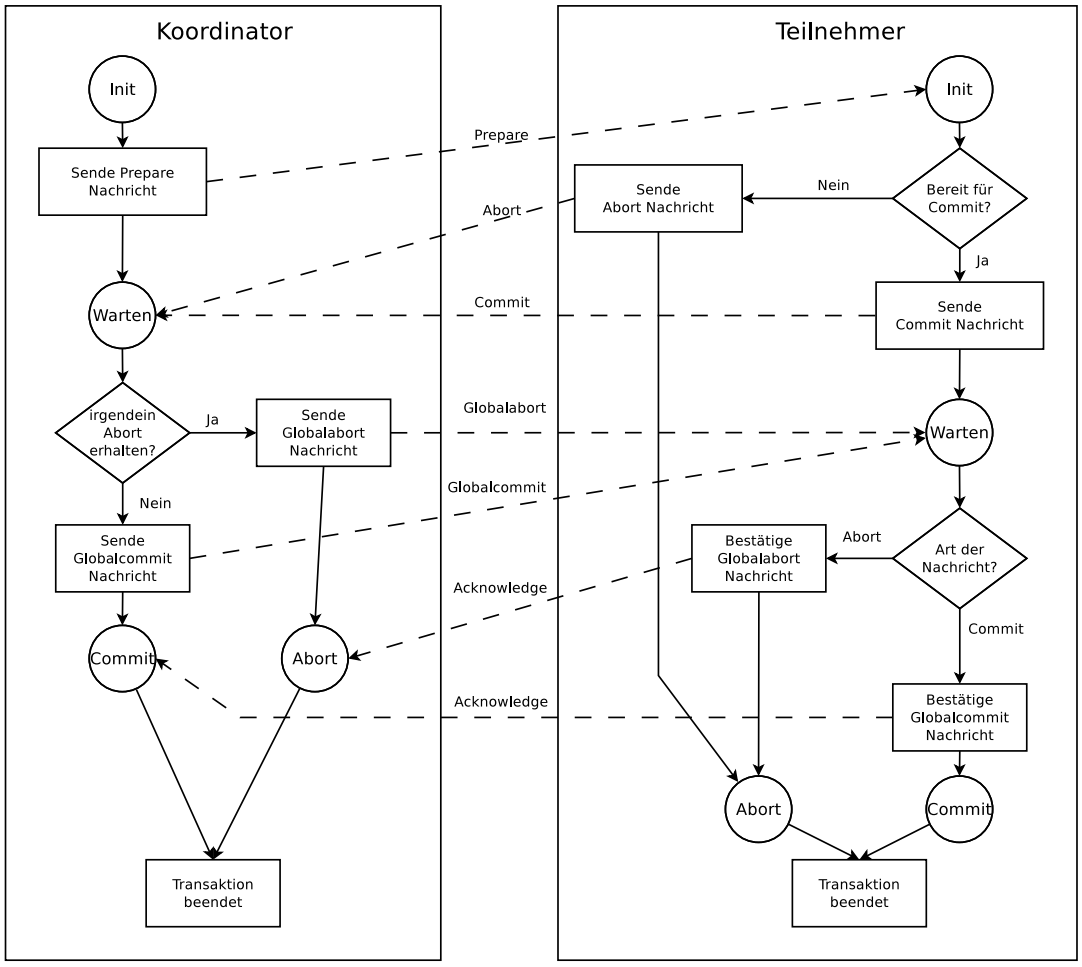


Abbildung 2.6: Ablauf des Zwei-Phasen-Commit-Protokolls [Dad96]

Fällt der Koordinator während der Commit-Behandlung aus und wird dieses durch zum Beispiel einen Timeout erkannt, wird ein neuer Koordinator gewählt. Um fortzufahren muss der neue Koordinator den *Commit*-Zustand der Teilnehmer erfragen. Wenn ein Teilnehmer den Erfolg oder den Abbruch der Transaktion meldet, wird dieses an alle Teilnehmer weitergegeben. Falls keiner einen Erfolg oder Abbruch meldet, aber mindestens einer sich im *PreCommit*-Zustand befindet, wird der Prozess mit Commit-Vorbereitung fortgesetzt. Ansonsten wird die Transaktion abgebrochen. Durch die dritte Phase ergibt sich ein höherer Kommunikationsaufwand. Das 3PC-Protokoll ist genauer in [Rah94] Abschnitt 7.2.5 beschrieben.

2.2.2 Quoren Strukturen

Um die Konsistenz der Replikate zu erhalten, müssen Schreiboperationen unter gegenseitigem Ausschluss stattfinden. Auch Leseoperationen dürfen sich mit Schreiboperationen zeitlich nicht überschneiden. Mehrere Leseoperationen hingegen können parallel ausgeführt werden, da sie die Replikate nicht modifizieren und die Konsistenz dadurch nicht gefährden.

Um diese Bedingungen zu erfüllen werden Quoren und Coterien genutzt [Sto06] [IST10]:

Definition 1 (Quorum) Eine *Quorenmenge* $\mathcal{Q}$ über alle Knoten $\mathcal{P}$ ist eine Menge von Untermengen unter $\mathcal{P}$ und es muss gelten:

$$\forall Q_1 \in \mathcal{Q} : Q_1 \neq \emptyset \wedge Q_1 \subseteq \mathcal{P} \quad \text{(Q1)}$$
$$\forall Q_1, Q_2 \in \mathcal{Q} : Q_1 \not\subseteq Q_2 \quad \text{(Q2)}$$

Definition 2 (Coterie) Eine Quorenmenge $\mathcal{Q}$ ist eine *Coterie* über alle Knoten $\mathcal{P}$ wenn gilt:

$$\forall Q_1, Q_2 \in \mathcal{Q} : Q_1 \cap Q_2 \neq \emptyset \quad \text{(C1)}$$

Je zwei Quoren aus der Quorenmenge $\mathcal{Q}$ die C1 erfüllen, haben mindestens einen gemeinsamen Knoten. Für den gegenseitigen Ausschluss von Lese- und Schreiboperationen muss jedes Quorum der Lesequorenmenge mit jedem Quorum der Schreibquorenmenge eine Coterie bilden. Dadurch wird zusätzlich gewährleistet, dass in jedem Lesequorum mindestens ein Knoten enthalten ist auf dem zuletzt eine Schreiboperation ausgeführt wurde und sich somit mindestens ein Knoten mit einem aktuellen Replikat darin befindet. Die Schreibquorenmenge muss ebenfalls zum gegenseitigen Ausschluss von Schreiboperationen eine Coterie bilden.

Jeder Knoten besitzt eine Stimme, die er abgeben kann, wenn eine Operation ausgeführt werden soll. Sie bleibt vergeben bis die Operation beendet ist. Gibt ein Knoten seine Stimme nicht ab, verweigert er damit die Zustimmung für die Durchführung der Operation. Um eine Operation ausführen zu können, muss der initiierende Knoten die Stimmen aller Knoten eines Quorums aus der zur Operation gehörenden Quorenmenge erhalten. Durch die Überschneidungseigenschaften einer Coterie besitzen zum Beispiel zwei Schreibquoren mindestens einen gemeinsamen Knoten, der seine Zustimmung nur einer Operation zur Zeit geben kann. Dadurch ist der gegenseitige Ausschluss gewährleistet.

Die Replikate besitzen eine Versionsnummer. Wird ein Quorum für eine Schreiboperation gefunden, werden die Daten auf alle Knoten des Quorums geschrieben und die Versionsnummer auf das Maximum der alten Versionsnummer plus eins gesetzt. Ein Lesequorum gibt das Replikat mit der höchsten Versionsnummer zurück, das dem aktuellsten Replikat entspricht [IST10].

2.2.3 Verfügbarkeit und Kosten

Durch replizieren der Daten an mehrere Orte sind die Daten öfter im Netzwerk vorhanden. Fällt ein Knoten aus, gibt es noch andere Knoten mit diesen Daten. Die Verfügbarkeit der Daten, auch als Fehler-Toleranz bezeichnet, eines Systems ergibt sich aus der Anzahl der Knoten die ausfallen können, ohne das System auszuschalten bzw. nutzlos zu machen [MRW97].

Eine Operation wird als verfügbar bezeichnet, wenn sie ausgeführt werden kann. Die Verfügbarkeit der Operationen auf den Daten wird durch die Quoren bestimmt. Je kleiner die Quoren sind, desto verfügbarer ist die Operation, da weniger Knoten benötigt werden und desto geringer sind auch die anfallenden Kosten. Je größer die Quoren werden, umso mehr Knoten mit Replikaten sind nötig, um die Operation auszuführen. Große Quoren sind somit kostenintensiver und verringern die Verfügbarkeit der Operationen. Da sich zum Beispiel Lese-

und Schreiboperationen überschneiden müssen, kann die Verfügbarkeit der einen Operation nur steigen, wenn die Verfügbarkeit der anderen dementsprechend sinkt [Her86]. Die Operationskosten ergeben sich in einem Netzwerk durch die Anzahl benötigter Nachrichten, die während der Operationen versendet werden müssen, wobei verschiedene Arten der Netzwerkverbindungen verschiedene Kosten verursachen [Sto06].

2.2.4 Partitionierung

Falls ein Verbindungsrechner (Gateway) zwischen zwei Subnetzen ausfällt, teilt sich das Netzwerk in zwei unabhängige Netzwerke. Eine Kommunikation zwischen diesen beiden Subnetzen ist dann nicht mehr möglich und es wird von einer Partitionierung des Netzwerks gesprochen.

Für die Erhaltung der Datenkonsistenz sind Partitionierungen problematisch, da theoretisch auf beiden Netzen gearbeitet werden kann. Wird zum Beispiel in dem einen Subnetz eine Schreiboperation ausgeführt und in dem Anderen eine Leseoperation, gibt die Leseoperation nicht das aktuellste Datum zurück. Replikationsstrategien müssen diese Inkonsistenz entweder verhindern oder auflösen [The02].

2.2.5 Klassische Strategien

Replikationsstrategien können in statisch oder dynamisch, optimistisch oder pessimistisch und syntaktisch oder semantisch eingeteilt werden.

Bei statischen Replikationsverfahren wird zu Beginn eine feste Anzahl von Knoten bestimmt, die am Replikationsverfahren teilnehmen. Die Anzahl kann während der Laufzeit nicht mehr angepasst werden, wodurch auf Knoten- und Kommunikationsausfälle nicht reagiert werden kann. Fallen mehr Rechner aus, als vom Verfahren toleriert werden kann, ist ein Zugriff auf die Daten mittels des Verfahrens nicht mehr möglich. Dynamische Verfahren lösen diese Problematik, indem die Quoren dynamisch nach Ausfällen auf die Anzahl der verfügbaren Knoten angepasst werden. Meistens wird die Anpassungsfähigkeit jedoch durch eine vorher festgelegte Obergrenze beschränkt. Die Anpassung bietet eine höhere Verfügbarkeit bei Knotenausfällen, jedoch muss ein höherer Koordinationsaufwand betrieben werden [Sto06][Dad96].

Der erhöhte Aufwand resultiert aus der Tatsache, dass ausgefallene Knoten nach ihrer Wiederherstellung die angepasste Strategie nicht kennen und nach der alten, ungültigen Strategie verfahren. Dadurch haben dynamische Verfahren nach der Anpassung und der darauf folgenden Wiederherstellung ausgefallener Knoten das Problem die Konsistenz zu wahren. Zur Vermeidung dieses Problems werden sogenannte Epochen eingesetzt. Eine Epoche beinhaltet alle am Verfahren teilnehmenden Knoten, die zu einer bestimmten Zeit verfügbar und verbunden sind. Eine Epochennummer dient zur Unterscheidung der verschiedenen Epochen. Unterscheidet sich die Anzahl der verfügbaren Knoten von der Anzahl der Knoten in der Epoche, wird versucht eine neue Epoche zu bilden. Dieser Epochenwechsel kann nur durchgeführt werden, wenn ein Schreibquorum der bisherigen und der neuen Epoche gebildet werden kann und alle enthaltenen Knoten dem Wechsel zustimmen. Der Epochenwechsel wird auf der Vereinigung der beiden Quorenmengen durchgeführt. Es können von Knoten der aktuellen Epoche Operationen ausgeführt werden. Befinden sich Knoten mit einer älteren Epoche in der Quorenmenge, muss

die Operation abgebrochen werden und der Knoten wird auf die aktuelle Epoche aktualisiert. Anschließend kann erneut versucht werden, die Operation durchzuführen [The02].

Bei optimistischen Verfahren wird angenommen, dass Inkonsistenzen der Daten die Ausnahme sind und selten auftreten. Deswegen versuchen die Verfahren nicht, Inkonsistenzen zu verhindern. Falls es doch einmal zu einer Inkonsistenz der Daten kommt, soll das Verfahren diese erkennen und auflösen, um wieder einen konsistenten Zustand herzustellen. Bei diesen Verfahren wird also mehr Wert auf eine hohe Verfügbarkeit und geringere Operationskosten zu Lasten der Datenkonsistenz gelegt. Pessimistische Verfahren hingegen schränken die Verfügbarkeit der Daten ein und besitzen höhere Operationskosten zu Gunsten der Konsistenz. Pessimistisch bedeutet in diesem Fall die Annahme, dass Inkonsistenzen auftreten, wann immer sie auftreten können [The02].

Bei einem syntaktischen Verfahren dient die 1-Kopien-Serialisierbarkeit als Korrektheitskriterium. Das bedeutet, eine parallele Ausführung von Operationen führt zum gleichen Ergebnis wie eine serielle Ausführung dieser Operationen auf nicht replizierten Daten. Semantische Verfahren nutzen daneben noch die Semantik der replizierten Daten als Korrektheitskriterium aus [The02][NHHT03].

Zusätzlich zu den genannten Einteilungen kann man Replikationsstrategien noch in strukturierte und unstrukturierte Verfahren einteilen. Strukturierte Verfahren nutzen dabei eine Struktur wie zum Beispiel Bäume oder Gitteranordnungen, um die Quoren zu bilden und so die Verfügbarkeit zu erhöhen und die Operationskosten zu senken [The02].

Nachfolgend werden einige pessimistisch-syntaktische Replikationsstrategien vorgestellt. Genauere Darstellungen der Verfahren und ihren Verfügbarkeiten sind unter anderem in [Dad96], [BS95] und [Sto06] zu finden.

Read One Write All-Verfahren

Bei dem Read One Write All-Verfahren werden Schreiboperationen immer auf allen Knoten durchgeführt. Für Leseoperationen wird immer genau ein Knoten benötigt. Dadurch sind die Lesekosten extrem günstig und die Leseoperation solange verfügbar, solange mindestens ein Knoten verfügbar ist. Schreiboperationen sind bei diesem Verfahren hingegen sehr teuer und nur möglich, wenn kein Knoten ausgefallen ist.

Quorum Consensus-Verfahren

Das Quorum Consensus-Verfahren benutzt das Konzept der Quoren, um das Verhältnis zwischen Lese- und Schreiboperationen flexibel zu gestalten, indem die benötigten Knoten einer Operation variieren, jedoch zu Beginn des Verfahrens festgelegt werden. Dabei wird ein neuer Wert nur auf Knoten des verwendeten Schreibquorums geschrieben. Leseoperationen lesen den Wert des Replikats von einem Knoten, der die aktuellste Version des Replikats im Lesequorum besitzt. Dafür müssen die Replikate einen Indikator besitzen, der die Aktualität beschreibt, um nicht ein veraltetes Datum zu lesen. Als Indikator können zum Beispiel Versionsnummern dienen. Aufgrund der Überschneidungseigenschaften von Lese- und Schreibquoren, die in Abschnitt 2.2.2

auf Seite 15 beschrieben sind, befindet sich im Lesequorum mindestens ein Knoten, der an der letzten Schreiboperation teilgenommen hat. Dadurch ist die Präsenz eines aktuellsten Replikats im Lesequorum gewährleistet. Wenn das Verfahren für jede Operation jeweils die einfache Mehrheit der Knoten benötigt, wird es auch als Majority Consensus Voting-Verfahren bezeichnet.

Das Quorum Consensus-Verfahren ist bei Schreiboperationen nicht so stark von Knotenausfällen betroffen, wie das Read One Write All, da die Schreibquorenmenge im Allgemeinen größer ist. Die Verfügbarkeit der Leseoperationen ist jedoch im Allgemeinen durch die größeren Lesequoren geringer. Das Read One Write All kann auch durch das Quorum Consensus-Verfahren nachgebildet werden.

Tree Quorum Protocol-Verfahren

Das Tree Quorum Protocol-Verfahren ordnet die Knoten in einer Baumstruktur an. Dabei werden je zwei Knoten durch eine gerichtete Kante verbunden. Der Quellknoten wird als Vaterknoten, der Zielknoten als Kindknoten bezeichnet. Der oberste Knoten der Struktur, der keinen Vaterknoten besitzt, ist der Wurzelknoten. Knoten, die keine Kinder besitzen, werden Blattknoten genannt. Alle Knoten mit Ausnahme des Wurzelknotens haben genau einen Vaterknoten. Der Grad d gibt die Anzahl der Kindknoten jedes Vaterknotens an. Ein Baum wird als komplett bezeichnet, wenn jeder Vaterknoten genau d Kindknoten besitzt. Alle Kindknoten des Wurzelknotens bilden die erste Ebene. Die Kindknoten der ersten Ebene bilden die zweite Ebene und so weiter. Die Höhe h eines Baumes ist die Anzahl der Ebenen.

Lesequoren werden aus der Mehrheit der Knoten einer Ebene des Baumes gebildet. Schreibquoren werden aus der Mehrheit von Knoten aus jeder Ebene des Baumes gebildet, wodurch die Überschneidungseigenschaften mit den Lesequoren gewährleistet sind.

Eine weitere Möglichkeit zur Quorenbildung ist wie folgt: Ein Lesequorum besteht aus dem Wurzelknoten. Ist dieser ausgefallen, wird die Mehrheit seiner Kindknoten benötigt. Sind in dieser Mehrheit Knoten ausgefallen, wird die Mehrheit der Kindknoten der vorher ausgewählten ausgefallenen Knoten benötigt. Schreibquoren bestehen aus dem Wurzelknoten und jeweils der Mehrheit der Kindknoten der ausgewählten Vaterknoten, bis herunter zu den Blattknoten.

Durch die Verwendung der Struktur wird die Anzahl der benötigten Knoten eines Quorums insbesondere für Schreibquoren verringert. Jedoch ergibt sich durch diese Vorgehensweise der Quorenbildung eine starke Abhängigkeit der Schreiboperationen vom Wurzelknoten, wodurch dieser besonders zuverlässig sein sollte.

Beispiel 1 Abbildung 2.7 auf der nächsten Seite zeigt das Tree Quorum Protocol-Verfahren mit sieben Knoten. Der Baum hat eine Höhe $h = 2$ und den Grad $d = 2$. Ein Lesequorum ist nach der zweiten vorgestellten Variante $\{K_0\}$ oder auch $\{K_1, K_2\}$. Sind die Knoten K_0 und K_1 ausgefallen, dann ist ein gültiges Lesequorum $\{K_2,K_3, K_4\}$.

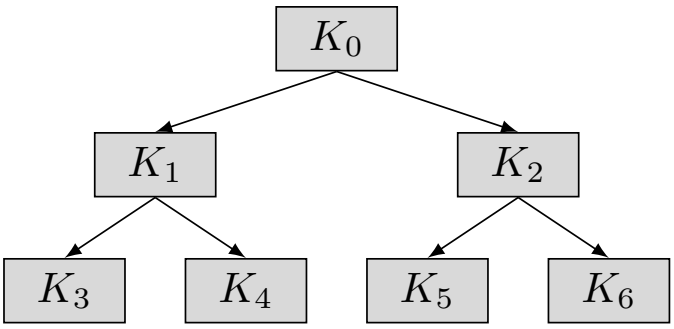


Abbildung 2.7: Tree Quorum Protocol mit 7 Knoten

Grid Protocol-Verfahren

Bei dem Grid Protocol-Verfahren werden die Knoten in einem $M \times N$ Gitter angeordnet, wobei M die Anzahl der Spalten und N die Anzahl der Zeilen ist. Durch diese Anordnung ist das Grid Protocol-Verfahren auf Knotenanzahlen beschränkt, die sich aus $M * N$ ergeben. Jeder Knoten in der Struktur ist mit allen Knoten seiner Nachbarspalten verbunden. Innerhalb einer Spalte ist jeder Knoten mit seinem direkten Nachbarn verbunden.

Ein Lesequorum besteht aus einem Knoten jeder Spalte des Gitters. Dies entspricht einem horizontalen Weg von links nach rechts durch das Gitter und wird *C-Cover* genannt. Alternativ kann ein Lesequorum auch aus einer kompletten Spalte des Gitters bestehen. Dies wird als *CC-Cover* bezeichnet. Ein Schreibquorum wird aus einem *C-Cover* und einem *CC-Cover* gebildet, wodurch die Überschneidungseigenschaften sichergestellt sind.

Durch die Gitteranordnung sind alle Knoten gleich gewichtet und das Problem unterschiedlich gewichteter Knoten, wie beim Tree Quorum Protocol-Verfahren, entfällt.

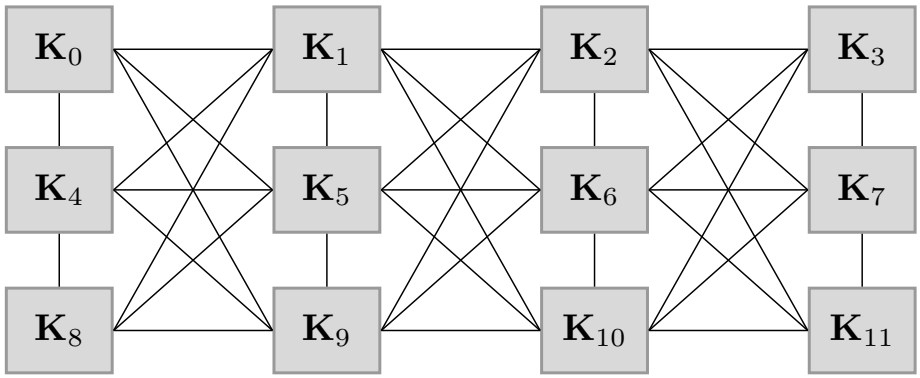


Abbildung 2.8: Grid-Protocol mit 12 Knoten

Beispiel 2 In Abbildung 2.8 ist ein Gitternetz mit 12 Knoten abgebildet. Ein Lesequorum ist nach dem Grid-Protocol $\{K_0, K_2, K_5, K_{11}\}$. Ein Schreibquorum wäre $\{K_1, K_4, K_5, K_6, K_7, K_9\}$.

Generalisierung statischer Verfahren (General Structured Voting)

Da jedes Replikationsverfahren für bestimmte Anwendungsszenarien Vor- und Nachteile gegenüber anderen Verfahren aufweist, muss das jeweils Beste für das gegebene Szenario ausgewählt und implementiert werden. Das General Structured Voting-Verfahren stellt ein Rahmenwerk zur Verfügung, mit dem statische Replikationsstrategien in einem einheitlichen Format beschrieben werden können. Dabei wird eine Trennung der Strategie eines Verfahrens von der Implementierung ermöglicht. Dadurch muss die Implementierung bei einer Anpassung oder dem Wechsel der Strategie nicht verändert werden, sondern nur die Strategiebeschreibung ausgetauscht werden.

Die Strategien werden durch einen azyklischen, gerichteten Graphen beschrieben, der *Voting-Struktur* genannt wird. Die Knoten der Voting-Struktur bestehen aus physikalischen und virtuellen Knoten. Ein physikalischer Knoten ist dabei eine reale Recheneinheit. Virtuelle Knoten dienen der Gruppierung physikalischer und virtueller Knoten, um eine Strategie zu erstellen.

Je zwei Knoten werden durch eine gerichtete Kante miteinander verbunden, wobei der Quellknoten als Vaterknoten und der Zielknoten als Kindknoten bezeichnet wird. Der oberste Knoten der Struktur, von dem aus die Struktur beginnt, ist der Wurzelknoten. Knoten die keine Kindknoten besitzen werden als Blattknoten bezeichnet. Jeder Knoten der Struktur besitzt eine Stimmenanzahl und jeweils eine Mindeststimmenanzahl für Lese- und Schreibquoren. Damit ein Knoten seine Stimme dem Vaterknoten geben kann, muss die Mindeststimmenanzahl der jeweiligen Operation durch die Summe der Stimmen der Kindknoten erreicht werden. Die Mindeststimmenanzahl der Operationen bei Blattknoten ist Null.

Durch die Priorisierung der Kanten durch eine Ganzzahl kann die Reihenfolge der Zugriffe auf die Kindknoten festgelegt werden. Je kleiner die Zahl ist, desto höher wird die Kante priorisiert. Ist keine Priorisierung angegeben, wird dies als niedrigste Priorität interpretiert. Bei mehreren Kanten mit der gleichen Priorität erfolgt der Zugriff innerhalb dieser Priorität zufällig. Einer Kante können unterschiedliche Prioritäten für Lese- und Schreiboperationen gegeben werden.

Die Quoren der Operationen werden ausgehend vom Wurzelknoten konstruiert. Dabei versucht jeweils der Vaterknoten, unter Beachtung der Prioritäten, die Zustimmung seiner Kindknoten für die auszuführende Operation einzuholen. Dies durchläuft den Baum bis zu den Blattknoten. Falls die Summe der Zustimmungen der Kindknoten die Mindeststimmenanzahl der Operation erreicht, kann der Vaterknoten seine Stimme abgeben. Erreicht der Wurzelknoten die geforderte Mindeststimmenanzahl und kann seine Stimme abgeben, kann die Operation mit allen zustimmenden Knoten als Quorum durchgeführt werden. Wird die Mindeststimmenanzahl des Wurzelknotens nicht erreicht, verweigert dieser die Stimmenabgabe und die Operation kann nicht durchgeführt werden.

Beispiel 3 In Abbildung 2.9 auf der nächsten Seite ist das Majority Consensus-Verfahren als Voting-Struktur zu sehen. V_0 ist ein virtueller Knoten, K_0 bis K_4 sind physikalische Knoten. Die obere Zahl an den Knoten gibt die Stimmenanzahl an, die der Knoten besitzt. Die unteren Zahlen sind die Mindeststimmenanzahlen für die Operationen, wobei die erste Zahl für ein Lesequorum und die zweite Zahl für ein Schreibquorum benötigt wird. V_0 braucht sowohl für ein Lesequorum als auch für ein Schreibquorum drei Stimmen seiner Kindknoten.

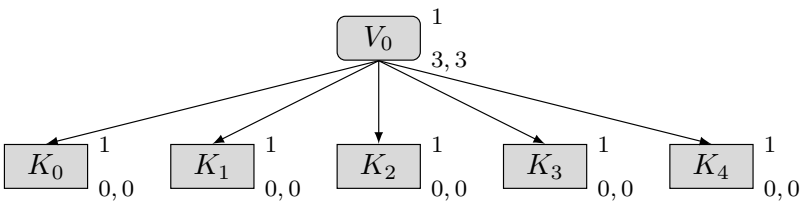


Abbildung 2.9: Majority Consensus-Verfahren mit 5 Knoten als Voting-Struktur

Dynamic Grid Protocol

Das Dynamic Grid Protocol erweitert das Grid Protocol um Dynamikeigenschaften. Wie bei der statischen Variante werden die Knoten in einem $M \times N$ Gitter angeordnet, wobei M die Anzahl der Spalten und N die Anzahl der Zeilen repräsentiert. Im Gegensatz zum Grid Protocol erlaubt das Dynamic Grid Protocol unbesetzte Stellen in der Gitterstruktur. Dadurch kann dieses Verfahren auch auf Knotenanzahlen die nicht durch $M * N$ gebildet werden können angewendet werden. N und M werden dabei so gewählt, dass es im Gitter minimal mehr Stellen als Knoten gibt. Die Stellen werden von links nach rechts und von oben nach unten mit den Knoten besetzt. Dadurch befinden sich die unbesetzten Stellen rechtsbündig in der unteren Zeile des Gitters und werden wie ausgefallene Knoten behandelt.

Bei der Bildung der Quoren dürfen keine unbesetzten Stellen in der Quorenmenge enthalten sein. Ein CC-Cover besteht somit nur aus allen belegten Stellen einer Spalte. Ein C-Cover wird aus jeweils einer besetzten Stelle jeder Spalte gebildet.

Zur Ausführung einer Leseoperation wird die Zustimmung eines C-Covers oder die Zustimmung eines CC-Covers des Gitters benötigt. Eine Schreiboperation kann bei der Zustimmung eines C-Covers und eines CC-Covers ausgeführt werden. Bei Knotenausfällen oder ihrer Wiederherstellung wird das Gitter an die neue Knotenanzahl automatisch angepasst.

Generalisierung dynamischer Verfahren (Dynamic General Structured Voting)

Das Dynamic General Structured Voting-Verfahren erweitert das General Structured Voting-Verfahren um Dynamikeigenschaften und trennt wie dieses, die Replikationsstrategie von der Implementierung. Durch die Trennung ist eine leichtere Anpassung der Strategie möglich. Verändert sich die Anzahl der Knoten durch Ausfälle oder der Wiederherstellung von Knoten, wird ein Epochenwechsel eingeleitet, bei dem die Struktur an die neuen Gegebenheiten angepasst wird. Hierbei dient eine vorher festgelegte Voting-Struktur, die Master Voting-Struktur, als Schablone. Aus dieser Schablone wird die angepasste Struktur, die Current Voting-Struktur, erstellt. Die physikalischen Knoten in der Master Voting-Struktur sind Platzhalter, denen bei der Erstellung der Current Voting-Struktur Knoten zugewiesen werden. Jede Current Voting-Struktur kann nur so viele Knoten enthalten, wie Platzhalter in der Master Voting-Struktur vorhanden sind, da diese statisch ist und zur Laufzeit nicht mehr verändert werden kann. Die Quoren werden bei diesem Verfahren genau wie im General Structured Voting-Verfahren gebildet.

Da für einige Replikationsstrategien bestimmte Knoten wichtiger sind als andere, sollten diese durch die zuverlässigsten der noch vorhandenen Knoten ersetzt werden. Ein sehr wichtiger Knoten ist zum Beispiel der Wurzelknoten des Tree Quorum Protocols. Dafür bietet das Dynamic General Structured Voting-Verfahren die Angabe einer Intaktwahrscheinlichkeit in dem Intervall (0,1] für jeden Knoten an. Je geringer der Wert, desto wahrscheinlicher ist ein Ausfall des Knotens.

Die Current Voting-Struktur wird während eines Epochenwechsels in drei Schritten aus der Master Voting-Struktur gebildet:

1. Alle an der neuen Epoche teilnehmenden Knoten werden auf die Platzhalter der Master Voting-Struktur verteilt. Dabei wird für jeden physikalischen und virtuellen Knoten ein Gewichtungswert bestimmt, der die Wichtigkeit des Knotens im Verfahren angibt. Je geringer der Wert ist, desto wichtiger ist der Knoten. Der Wert des Vaterknotens muss immer höher sein, als die jeweiligen Werte seiner Kindknoten. Des Weiteren muss der Wert eines Knotens höher bzw. gleich dem Wert seines Bruderknotens sein, wenn die Kante vom Vaterknoten zu diesem Knoten eine höhere bzw. gleiche Priorität als zu seinem Bruderknoten aufweist.

   Die Knoten werden absteigend nach ihrer Intaktwahrscheinlichkeit sortiert. Alle Knoten mit gleicher Wahrscheinlichkeit werden nach einem eindeutigen Merkmal wie zum Beispiel dem Namen absteigend sortiert. Dadurch ergibt sich eine eindeutige Reihenfolge. Alle Platzhalterknoten der Master Voting-Struktur werden absteigend nach ihrem Gewichtungswert sortiert. Diese beiden Listen werden dann zusammengefügt, indem jeder i-te Knoten der einen Liste dem i-ten Knoten der anderen Liste zugewiesen wird. Der wichtigste Knoten wird dabei automatisch mit dem Zuverlässigsten besetzt und jeder Platzhalterknoten, dem kein Knoten zugewiesen wurde, folgen nur Platzhalter, denen ebenfalls kein Knoten zugewiesen wurde.

2. Konnten nicht allen Platzhalterknoten mit gleichem Gewichtungswert Knoten zugeordnet werden, können Knoten wichtiger werden, als von der verwendeten Strategie vorgesehen. Dies äußert sich zum Beispiel dadurch, dass ein Knoten drei Kindknoten besitzt und ein anderer Knoten, mit gleicher Gewichtung, nur einen Kindknoten. Um dies zu beheben wird ein horizontaler Ausgleich durchgeführt, bei dem die Knoten gleichmäßig auf die Platzhalterknoten gleicher Gewichtung verteilt werden. Da alle Platzhalterknoten mit dem gleichen Gewichtungswert auf einer Ebene liegen, ist dies zulässig. Wenn noch Platzhalter ohne Zuweisung vorhanden sind, werden diese entfernt. Virtuelle Knoten, die keinen oder nur einen Kindknoten haben, werden ebenfalls entfernt beziehungsweise durch den Kindknoten ersetzt.

3. Durch Umstrukturierung kann es vorkommen, dass die Kindknoten die Mindeststimmenanzahl für die Operationen nicht mehr erfüllen können. In diesem Fall werden neue Mindeststimmenanzahlen für den Vaterknoten berechnet, die der Strategie entsprechen und durch die Kindknoten erfüllbar sind. Zuletzt wird die neue Strategie den Knoten der Vereinigungsmenge, der bei dem Epochenwechsel benötigen Schreibquoren, bekannt gegeben. Für eine genaue Beschreibung des Epochenwechsels sei hier auf [Sto06] und [The02] verwiesen.

In [Sto06] wird eine Erweiterung des Dynamic General Structured Voting eingeführt, die ein Hinzufügen und Entfernen von Knoten während der Laufzeit ermöglicht, um eine flexible Anzahl von Replikaten verwalten zu können. Außerdem wird ein Mechanismus zur Benutzung verschiedener Strategien für verschiedene Knotenanzahlen vorgestellt, der somit zu inhomogenen Strategien führt.

2.2.6 Gerneral Quorum Consensus

Das General Quorum Consensus-Verfahren [Her86] verallgemeinert das in Abschnitt 2.2.5 auf Seite 18 beschriebene Quorum Consensus-Verfahren. Indem spezifische Eigenschaften von Daten systematisch ausgenutzt werden, wird eine höhere Verfügbarkeit als in konventionellen Klassifikationen von Lese- / Schreiboperationen erreicht. Das Verfahren ist somit in die pessimistisch-semantischen Verfahren einzuordnen. Auf den Daten können beliebige Operationen modelliert werden, die jedoch atomar sein müssen. Siehe hierzu auch Abschnitt 2.2.1 auf Seite 13. Datencontainer werden als Objekte bezeichnet und sind abstrakte Datentypen. Jedes Objekt bietet eine Menge von Operationen, um das Objekt zu erstellen und zu manipulieren. Ein Operationspaar von Aufruf und Antwort ist ein Ereignis. Eine Sequenz von Ereignissen ergibt den Zustand eines Objekts und wird Historie genannt. Jede Operation benötigt eine gewisse Anzahl von Replikaten, die durch Quoren dargestellt werden. Die Quoren werden in initiale und finale Quoren eingeteilt, wobei der Aufruf auf einem initialen Quorum und die Antwort auf einem finalen Quorum ausgeführt wird. Somit unterteilt sich jedes Ereignis in ein initiales und ein finales Quorum. Jedes Ereignis besteht aus einem Timestamp, dessen Wert aus einem total geordneten Bereich kommt und aus der Art der Operation mit dessen Parametern. Der Objektzustand wird durch ein Log repräsentiert, in dem die Ereignisse festgehalten werden. Jedes Replikat ist eine partielle Historie der Ereignisse.

Eine Operation wird wie folgt durchgeführt [The02] [Her86]:

1. Replikate eines initialen Quorums im Lesemodus sperren.
2. Vereinigung aller partiellen Historien der Replikate des initialen Quorums. Es wird nach dem Timestamp sortiert und Duplikate werden gelöscht. Die erstellte Liste wird View genannt und durch Auswerten des Views wird der aktuelle Zustand des replizierten Objekts gebildet.
3. Die geplanten Operationen auf dem aktuellen Zustand des Objekts ausführen.
4. Die Operationen mit einem Timestamp an den View anhängen.
5. Replikate eines finalen Quorums im Schreibmodus sperren.
6. Historie der Replikate des finalen Quorums mit dem View aktualisieren. Die Einträge werden nach dem Timestamp sortiert und Duplikate gelöscht.
7. Lese- und Schreibsperren aufheben.

Benötigt eine Operation OP_2 Informationen über eine früher ausgeführte Operation OP_1 muss für die initialen Quoren iq und die finalen Quoren fq gelten [The02]:

$$iq_{OP_2} \cap fq_{OP_1} \neq 0$$

Da die partiellen Historien der Replikate stark wachsen, sollten diese regelmäßig durch einen aktuellen Objektwert ersetzt werden.

Beispiel 4 Gegeben sei ein Objekt x mit dem Datentyp Warteschlange. Das Objekt stellt die Operationen $enq(x,a)$ und $a = deq(x)$ zur Verfügung. *enq* reiht dabei ein Element hinten in die Warteschlange ein und *deq* entfernt das vorderste Element der Warteschlange. Eine *deq*-Operation steht also in einer Abhängigkeit von einer *enq*-Operation, da diese Kenntnis von den eingetragenen Elementen haben muss. Weiter steht eine *deq*-Operation auch mit anderen *deq*-Operationen in Abhängigkeit, da bekannt sein muss, welche Elemente schon entfernt wurden.

Bei einer Teilnahme von vier Knoten an diesem Verfahren ergibt sich zum Beispiel für das initiale Quorum der *enq*-Operation die Größe $iq_{enq} = 0$ und für das finale Quorum die Größe $fq_{enq} = 2$. Dann muss das initiale Quorum der *deq*-Operation eine Größe von $iq_{deq} = 3$ und das finale Quorum eine Größe von $fq_{deq} = 2$ haben.

2.2.7 Probabilistische Verfahren

Pessimistische Replikationsstrategien gehen davon aus, dass keine Operation zu inkonsistenten Replikaten führt. Dadurch ist die Verfügbarkeit der Operationen dieser Verfahren stark begrenzt. Probabilistische Quoren Systeme opfern einen gewissen Grad der Konsistenz. Die Konsistenzbedingungen werden so abgeschwächt, dass sich die Quoren nur mit einer sehr hohen Wahrscheinlichkeit überschneiden.

Durch diese Wahrscheinlichkeit ergibt sich die Möglichkeit, dass zwei Operationen hintereinander ausgeführt werden, dessen Quoren sich nicht überschneiden und dadurch die Konsistenz gefährden. Die Operationsverfügbarkeit wird dabei jedoch drastisch verbessert und ergibt somit einen Kompromiss zwischen Datenkonsistenz und Operationsverfügbarkeit. Je höher die Konsistenzanforderungen sind, desto geringer ist die Operationsverfügbarkeit und anders herum. [MRW97]

Trotz der nicht vollständig gewährleisteten Überschneidung der Knotenmengen, wie sie in der Definition in Abschnitt 2.2.2 auf Seite 15 angegeben ist, werden diese Mengen folgend weiterhin Quoren genannt. Quoren mit den Überschneidungseigenschaften werden in diesem Abschnitt als strikte Quoren bezeichnet.

Probabilistische Quorensysteme sind somit am Besten geeignet, wenn die Verfügbarkeit von Operationen wichtiger ist, als die Konsistenz der Daten. Dies könnte bei reaktionskritischen Systemen der Fall sein, denen ein orientierender Wert ausreicht und bei dem lange Antwortzeiten fehlerträchtiger sind als falsche Werte [IST10].

[IST10] beschreibt zwei probabilistische Quoren-Konstruktions-Algorithmen, die einen Kompromiss zwischen Datenkonsistenz und Operationsverfügbarkeit darstellen. Dabei ist die Grundidee Operationen zuerst mit strikten Quoren durchzuführen. Erst wenn diese nicht verfügbar sind,

wird auf die probabilistischen Quoren mit den abgeschwächten Konsistenzbedingungen zurückgegriffen. Sind diese Quoren auch nicht verfügbar, kann die Operation nicht durchgeführt werden.

Die Konstruktionen sehen vor, dass aus einem strikten Quorensystem ein probilistisches Quorensystem generiert wird und mit dem Strikten vereinigt wird. Dabei erhalten die strikten Quoren eine höhere Priorität. Haben die generierten Quoren eine kleine Kardinalität, führt dies zu einer hohen Operationsverfügbarkeit und zu einer niedrigen Konsistenz.

Zwei Kriterien sind für die Konstruktionen vorgegeben:

- Die Quorengrößen im probabilistischen System müssen kleiner als im strikten System sein
- Die Quorengrößen im probabilistischen System müssen groß genug sein, um vorgegebene Überschneidungswahrscheinlichkeiten sicherzustellen.

Die genauen Algorithmen der beiden Konstruktionen, Minimal Quorums-Based Construction und All Quorums-Based Construction, können in Abschnitt 4.1 und 4.2 in [IST10] nachgeschlagen werden.

Kapitel 3

Konzept für Replikationsstrategien

Das Kapitel beschreibt das theoretische Konzept des zu entwickelnden Rahmenwerkes und beginnt mit dessen Anforderungen. Im darauf folgenden Abschnitt werden die Voting-Strukturen des General Structured Voting-Verfahrens angepasst und danach eine neue Struktur, die Gruppenstruktur, für die Replikationsstrategien vorgestellt. Diese Struktur soll die Reduzierung von Energiekosten ermöglichen. Die letzten beiden Abschnitte beschreiben die Erstellung von Quoren für Operationen durch die Gruppen- und Voting-Strukturen.

3.1 Anforderungen

Replikationsstrategien müssen im Rahmenwerk leicht austauschbar sein, um die verschiedenen Strategien effizient untersuchen zu können. Deswegen wird auf das General Structured Voting-Verfahren, das in Abschnitt 2.2.5 auf Seite 21 erklärt wird, zurückgegriffen. Das Verfahren trennt die Implementierung von der eigentlichen Replikationsstrategie, wodurch ein leichter Austausch ermöglicht wird. Ohne diese Trennung müsste jede Strategie eigenständig implementiert werden, wodurch sich die Untersuchung verschiedener Strategien erheblich verlängert.

Neben der leichten Austauschbarkeit soll das Rahmenwerk die Grundlagen des General Quorum Consensus-Verfahrens bereitstellen, das in Abschnitt 2.2.6 auf Seite 24 beschrieben ist. Dadurch wird das Testen von Verfahren mit abstrakten Datentypen und verschiedenen Operationen zusätzlich zu den klassischen Verfahren mit Schreib-/Lese-Operationen ermöglicht.

Da ein Sensornetz häufig aus Knoten mit gleichen Sensoren besteht, wird aus den Messdaten meistens ein Durchschnittswert gebildet. Da Messdatenausreißer durch diese Mittelung der Werte bei großen Mengen nicht ins Gewicht fallen und somit auch Fehlmessungen toleriert werden können, soll das Rahmenwerk probabilistische Verfahren für die Verteilung der Daten unterstützen.

3.2 Erweiterte Voting-Strukturen

Um die gestellten Anforderungen an das Rahmenwerk zu erfüllen, werden die Voting-Strukturen des General Structured Voting-Verfahrens übernommen und angepasst. Voting-Strukturen bestehen aus physikalischen und virtuellen Knoten. Physikalische Knoten stellen dabei die

realen Sensorknoten dar. Virtuelle Knoten dienen zur Gruppierung und Strukturierung, um die Replikationsstrategie als azyklischen Graphen zu bilden. Dafür werden je zwei Knoten durch eine gerichtete Kante miteinander verbunden. Der Quellknoten wird als Vaterknoten und der Zielknoten als Kindknoten bezeichnet. Der Knoten, von dem aus die Struktur beginnt, ist der Wurzelknoten. Knoten, die keine Kindknoten besitzen, heißen Blattknoten. Blattknoten sind immer physikalische Knoten.

Die Reihenfolge der Zugriffe auf die Kindknoten kann durch eine Priorisierung der Kanten erfolgen, die durch eine Ganzzahl festgelegt wird. Je kleiner diese Zahl ist, desto höher ist die Kante priorisiert. Kanten ohne eine Angabe erhalten automatisch die niedrigste Priorität. Besitzen mehrere Kanten die gleiche Priorität, wird zufällig eine dieser Kanten ausgewählt.

Jeder Knoten der Struktur bekommt eine Stimmenanzahl, die er dem Vaterknoten übergeben kann. Außerdem enthält jeder Knoten ein Zweier-Tupel von Mindeststimmenanzahlen für jede Operation. Das erweitert die Strukturen des General Structured Voting-Verfahrens, die nur eine Mindeststimmenanzahl für jeweils Lese- und Schreiboperationen enthalten, um das General Quorum Consensus-Verfahren. Für die Tupel muss eine eindeutige Reihenfolge festgelegt sein, die bei jedem Knoten eingehalten werden muss, um sie den Operationen zuordnen zu können.

Der erste Wert des Zweier-Tupels gibt die benötigten Stimmen für ein initiales Quorum und der zweite Wert für ein finales Quorum an. Damit ein Knoten seine Stimme dem Vaterknoten übergeben kann, muss die Mindeststimmenanzahl für die jeweilige Operation durch die Summe der Stimmen der Kindknoten erreicht werden.

Die Quoren werden ausgehend vom Wurzelknoten konstruiert. Dabei versucht jeweils der Vaterknoten die Zustimmung seiner Kindknoten unter Beachtung der Prioritäten zu erhalten. Die Abfrage wird bis zu den Blattknoten durchgeführt. Erreicht ein Vaterknoten die Mindeststimmenanzahl, kann er seine Stimme seinem Vaterknoten geben. Wird die Mindeststimmenanzahl jedoch nicht erreicht, verweigert der Vaterknoten die Abgabe seiner Stimme. Sobald der Wurzelknoten die Mindeststimmenanzahl für eine Operation erreicht, kann diese mit allen zustimmenden Knoten, die dann das Quorum bilden, durchgeführt werden.

Beispiel 5 Ein Objekt x mit dem Datentyp Warteschlange soll auf vier Knoten repliziert werden. Das Objekt stellt die Optionen $enq(x,a)$ zum Einreihen des Wertes a in die Warteschlange und $a = deq(x)$ zum Entfernen des ersten Elements der Warteschlange zur Verfügung. *enq* und *deq* Operationen und zwei *deq* Operationen stehen dabei in Abhängigkeit zueinander, da den *deq* Operationen bekannt sein muss, welche Elemente eingefügt und entfernt wurden. *enq*-Operationen brauchen diese Informationen nicht.

Dadurch ergibt sich für *enq*-Operationen zum Beispiel eine Größe der initialen Quoren von $iq_{enq} = 0$ und der finalen Quoren von $fq_{enq} = 2$. Die Größe der initialen Quoren der *deq*-Operation müssen dann $iq_{deq} = 3$ und die der finalen Quoren $fq_{deq} = 2$ sein.

In Abbildung 3.1 ist eine erweiterte Voting-Struktur zur Bildung dieser Quoren angegeben. Die obere Zahl an den Knoten ist die Stimme und die unteren Zweier-Tupel sind die Mindeststimmenanzahlen für die Operationen. Dabei gibt das erste Tupel die Mindeststimmenanzahl für die *enq*-Operation und das zweite Tupel die Mindeststimmenanzahl für die *deq*-Operation an.

Um eine *enq*-Operation auszuführen, muss zuerst ein initiales Quorum gebildet werden. Der Wurzelknoten benötigt in diesem Fall null Stimmen seiner Kindknoten und kann somit direkt

seine Stimme abgeben. Dies führt zum ersten Teil der Operationsausführung. Siehe hierzu auch das General Quorum Consensus-Verfahren in Abschnitt 2.2.6 auf Seite 24. Als Zweites benötigt die *enq*-Operation ein finales Quorum, das aus zwei Kindknoten des Wurzelknotens bestehen muss. Ein gültiges finales Quorum wäre somit zum Beispiel $\{K_0, K_2\}$.

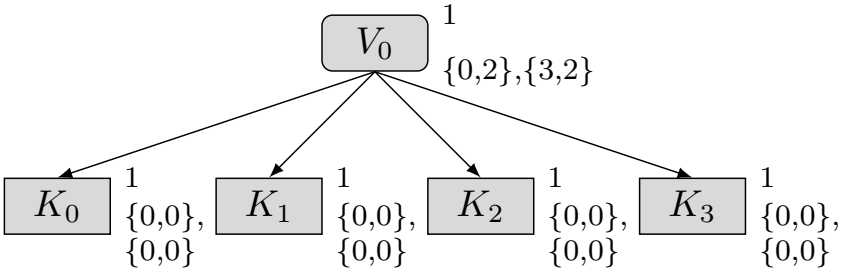


Abbildung 3.1: Angepasste Voting-Struktur

Probabilistische Quoren-Systeme können durch die Voting-Strukturen dargestellt werden, in dem die Mindeststimmenanzahlen so angepasst werden, dass die Überschneidungseigenschaften nicht mehr in vollem Maße und nur noch nach Vorgaben gewährleistet sind. Die Abbildung 3.2 zeigt diesen Ansatz. In diesem Beispiel wurde das Majority Consensus Voting-Verfahren aus Abschnitt 2.2.5 auf Seite 18 verwendet und die Quorengröße um eins verringert. Eine Kombination aus strikten und probabilistischen Systemen, wie bei [IST10], kann durch zwei Kindknoten des Wurzelknotens mit Priorisierung erfolgen. Dabei bekommt der Kindknoten, unter dem die strikten Quoren gebildet werden, eine höhere Priorität.

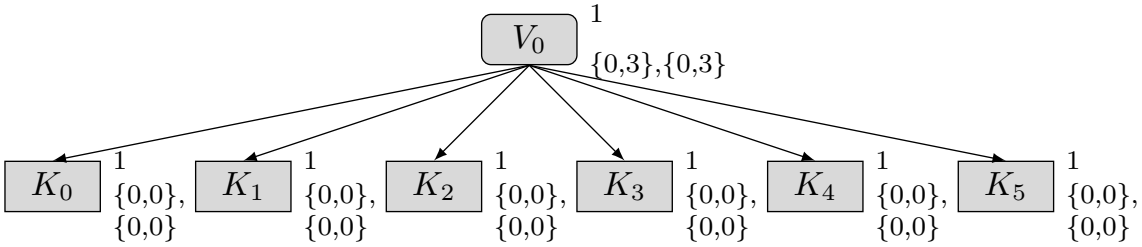


Abbildung 3.2: Angepasste probabilistische Voting-Struktur

Die erweiterten Voting-Strukturen beinhalten keine Angaben, mit denen der Energieverbrauch beachtet werden kann. In Sensornetzen entstehen jedoch nicht nur Kosten durch die Anzahl der Nachrichten die versendet werden müssen, sondern auch durch die Entfernung der Knoten zueinander. Um diesen Aspekt in die Voting-Strukturen zu integrieren, wurde eine zusätzliche Struktur entwickelt, die über den Voting-Strukturen liegt. Diese Strukturen werden im nächsten Abschnitt beschrieben.

3.3 Gruppenstruktur

Das Replizieren von Daten verursacht Kosten durch den Austausch von Nachrichten. In einem Sensornetzwerk unterscheiden sich die Kosten für die einzelnen Nachrichten durch die Entfernung vom Sender zum Empfänger. Abbildung 3.3 zeigt einen Knoten K_0 der als Beispiel zwei Sendeentfernungen besitzt. Die innere gestrichelte Linie zeigt die erste Reichweite des Knotens. Die äußere gestrichelte Linie die zweite Reichweite, für die der Knoten mehr Energie aufbringen muss.

Soll von K_0 eine Nachricht versendet werden, ist der Energieverbrauch für die Zustellung an K_1, K_2 oder K_4 geringer, als an die restlichen weiter entfernten Knoten. Dies ist nicht nur bei einer Single-Hop Kommunikation der Fall, sondern auch bei einer Multi-Hop Kommunikation. Um eine Nachricht an K_7 über K_4 zu schicken, verbraucht der Knoten K_0 zwar nur Energie für die erste Reichweite, jedoch verbraucht zusätzlich K_4 Energie.

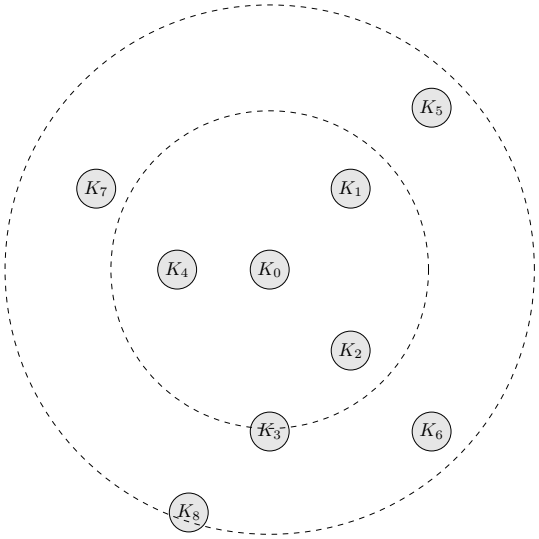


Abbildung 3.3: Sensornetz mit verschiedenen Sendereichweiten von K_0

3.3.1 Erste Ansätze

Um die verschiedenen Kosten der Nachrichten zu berücksichtigen, muss die vorher erweiterte Voting-Struktur weiter angepasst werden. Die erste Idee ist, die Knoten, die bei der Kommunikation weniger Kosten verursachen, priorisiert zu behandeln. Dies kann durch eine einfache Kantengewichtung, wie sie schon in den Strukturen vorgegeben ist, realisiert werden. Dadurch ergeben sich jedoch zwei Probleme:

1. Wird davon ausgegangen, dass nur ein Knoten seine Daten repliziert, würden immer nur seine unmittelbaren Nachbarn durch die Priorisierung bevorzugt. Dadurch entsteht ein Ungleichgewicht bei den Energiekosten im Netzwerk und ein Teil des Netzes würde vermutlich durch Energiemangel vorher ausfallen.

2. Die Daten werden nur auf einer kleinen Fläche des Netzes verteilt. Tritt ein Flächenschaden im Netzwerk auf, durch zum Beispiel herabfallende Steine, könnten eher alle Knoten mit den Replikaten des einen Sensorknotens zerstört werden. Dadurch wären alle gesammelten Daten verloren.

Es ist somit von Vorteil die Daten weitläufiger auf das Netzwerk zu verteilen. Einerseits um einen besseren Energieausgleich zu erzielen, andererseits um die Replikate auf weiter Fläche zu streuen und so Datenverlusten vorzubeugen. Dennoch sollten hierbei bevorzugt kostengünstige Knoten angesprochen werden.

Um diese Vorgaben zu erzielen, wurden im zweiten Ansatz die Sensorknoten durch die Entfernung gewichtet und an den Kanten der Voting-Strukturen statt Prioritäten eine Reihenfolge der Benutzung festgelegt. Diese Reihenfolge stellt eine Art Abzählreim dar, der es zum Beispiel ermöglicht drei Mal Quoren mit geringen Kosten, dann zweimal mit mittleren und einmal mit hohen Kosten zu bilden. Diese Reihenfolge wird dann permanent wiederholt.

Die Konstruktion der Strukturen erschwert sich jedoch durch diese Vorgehensweise und gleichzeitig geht die Eigenschaft der Kantenpriorisierung verloren. Deswegen wurden in der endgültigen Struktur die Abzählreime herausgezogen und über die Voting-Struktur gelegt. Der nachfolgende Abschnitt beschreibt diese Struktur, die im Folgenden Gruppenstruktur genannt wird.

3.3.2 Die Struktur

Eine Gruppenstruktur besteht aus Gruppenknoten, die durch gerichtete Kanten miteinander verbunden sind. Ein Token markiert die zuletzt in einem finalen Quorum verwendete Gruppe, von der aus die nächste Gruppe zur Bildung der Quoren bestimmt wird. Es können nur die direkten Nachfolger dieses Knotens verwendet werden. Die Zugriffsreihenfolge kann durch Priorisierung der Kanten mit einer Ganzzahl bestimmt werden. Je kleiner die Zahl ist, desto höher ist die Priorität der Kante. Bei mehreren Kanten mit gleicher Priorität wird zufällig zwischen ihnen ausgewählt. Kanten, denen keine Priorität zugewiesen wurde, haben die geringste Priorität in der Struktur.

Die einzelnen Gruppenknoten können eine Gruppenstruktur oder eine erweiterte Voting-Struktur als untergeordnete Strukturen beinhalten, wobei jede Gruppe irgendwann mit einer Voting-Struktur abgeschlossen werden muss. In Abbildung 3.4 auf der nächsten Seite enthält der Gruppenknoten G_1 beispielsweise erneut eine Gruppenstruktur als untergeordnete Struktur. Die Gruppen dieser Struktur enthalten entweder wieder eine Gruppenstruktur oder enden mit einer Voting-Struktur. Die Gruppe G_2 in der Abbildung verweist im Gegensatz zu G_1 direkt auf eine erweiterte Voting-Struktur.

Durch die Gruppenstrukturen können mehr als eine erweiterte Voting-Struktur in der Replikationsstrategie vorhanden sein, weswegen das nachfolgende Konzept der Rückverweise entwickelt wurde.

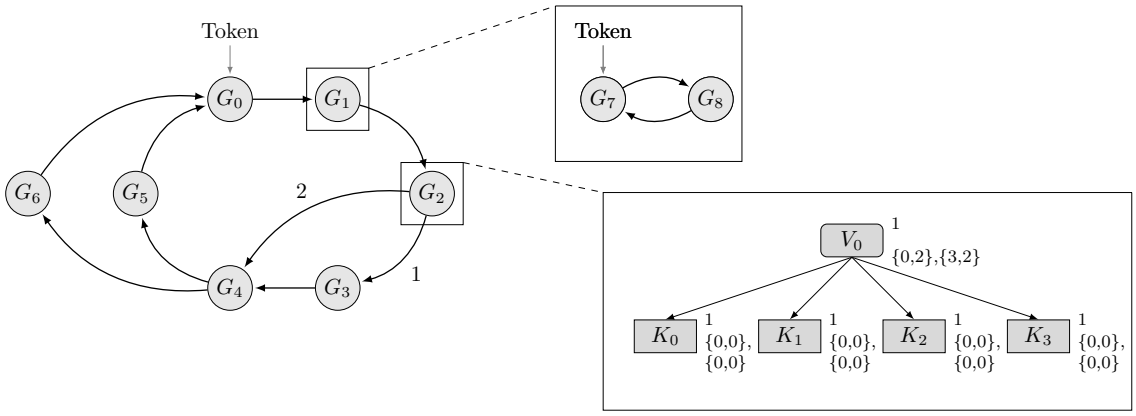


Abbildung 3.4: Gruppenstruktur

3.3.3 Rückverweise

Bei der Verwendung von mehreren Voting-Strukturen können Lücken in den Historien der Replikate entstehen, da jede Voting-Struktur separat für sich arbeitet und es keine Gewährleistung für Überschneidungen der Voting-Strukturen gibt. Wird beispielsweise ein Wert x in einer Struktur a in eine Warteschlange eingereiht, kann eine Struktur b den Wert x durch den Aufbau eines Views mit den partiellen Historien nicht sehen, wenn sich Struktur a und b nicht überschneiden. Für die Erstellung eines Views sei hier auf Abschnitt 2.2.6 auf Seite 24 verwiesen.

Somit wird ein Objekt nicht vollständig innerhalb einer Voting-Struktur repliziert, sondern nur Teile der Historie. Um den aktuellen Objektzustand wieder herzustellen, müssen die Historienteile aller Voting-Strukturen der Strategie zusammengefügt werden.

Damit in der zuletzt verwendeten Voting-Struktur alle Teile einer Historie gespeichert werden und so immer das vollständige und aktuelle Objekt erstellt werden kann, sind Überschneidungseigenschaften zwischen den Voting-Strukturen erforderlich.

Definition 3 Seien zwei Voting-Strukturen S_1 und S_2 und ein Objekt mit einer Schreiboperation OPS und einer Leseoperation OPL gegeben. Wird durch die Gruppenstrukturen die Voting-Struktur S_2 nach S_1 benutzt, muss für die Bildung eines vollständig enthaltenen Views in der Struktur S_2 gelten:

$$fq_{OPS_{S_1}} \cap iq_{OPS_{S_2}} \neq \emptyset$$

Das initiale Quorum der Operation OPS auf der Struktur S_2 muss das zuletzt verwendete finale Quorum der Operation OPS der Struktur S_1 schneiden. Dadurch enthält das initiale Quorum von OPS_{S_2} mindestens einen Knoten aus der Struktur S_1, auf dem das Objekt zuletzt verändert

wurde. Dadurch kann ein vollständiger View erstellt werden, auf dem OPS ausgeführt wird. Danach wird der View auf einem finalen Quorum in der Struktur S_2 verteilt.

Stehen OPS und OPL in Abhängigkeit zueinander, kann die Überschneidungseigenschaft zum Beispiel auch gewährleistet werden durch:

Definition 4 (Rückverweis)

$$iq_{OPS_{S_2}} = iq_{OPL_{S_1}}$$

Das initiale Quorum einer Operation OPS auf der Struktur S_2 besteht aus einem initialen Quorum der Operation OPL auf der Struktur S_1. Im Allgemeinen muss das initiale Quorum jedoch so gewählt werden, dass jede Änderung jeder Operation aus der vorhergegangenen Voting-Struktur enthalten ist.

Der Rückverweis muss immer gesetzt werden, sobald eine Operation Kenntnis von anderen Operationen aus einer vorher benutzten Voting-Struktur benötigt. In der Voting-Struktur kann ein Rückverweis anstelle der Mindeststimmenanzahl für die Operation an den Wurzelknoten geschrieben werden. Dafür wird ein R plus eine Ganzzahl angegeben. Die Zahl wird an den Mindeststimmenanzahlen des Wurzelknotens der zuletzt benutzten Struktur von links nach rechts abgezählt. Bei einer Mindeststimmenanzahl der Form $\{initialesQuorum_{OP_1}, finalesQuorum_{OP_1}\}$, $\{initialesQuorum_{OP_2}, finalesQuorum_{OP_2}\}$ bedeutet $R4$, dass ein finales Quorum der Operation OP_2 genutzt werden soll.

3.4 Daten verteilen

Zum Verteilen der Daten muss zunächst ein initiales Quorum gebildet werden. Dazu wird die Voting-Struktur der zuletzt verwendeten Gruppe mit dem Rückverweis der Operation benutzt. Aus dieser Struktur wird das initiale Quorum, wie in Abschnitt 3.2 auf Seite 27 beschrieben, gebildet.

Nachdem die Operation mit einem initialen Quorum durchgeführt wurde, muss sie mit einem finalen Quorum abgeschlossen werden. Dafür wird ein Gruppenknoten ausgewählt, der ein Nachfolger der zuletzt verwendeten Gruppe der obersten Gruppenstruktur ist. Enthält diese Gruppe als Unterstruktur eine Gruppenstruktur, wird in dieser wieder von der zuletzt verwendeten Gruppe dieser Struktur die nächste Gruppe ausgewählt. Dies wird solange fortgeführt, bis eine erweiterte Voting-Struktur erreicht ist, aus der dann das finale Quorum gebildet wird.

Kann aus der Struktur kein Quorum gebildet werden, wird zurück in die übergeordnete Gruppenstruktur gewechselt und von dort aus der nächste Gruppenknoten ausgewählt. Die Strukturen werden dann erneut bis herunter zu einer Voting-Struktur durchgegangen. Wurden in der übergeordneten Struktur alle möglichen Nachfolgeknoten benutzt, wird wieder in eine Strukturebene höher gewechselt. Sobald alle möglichen Nachfolger der obersten Ebene zu keinem gültigen Quorum geführt haben, kann die Operation nicht ausgeführt werden.

Beispiel 6 In Abbildung 3.5 ist die Erstellung eines Quorums zum Verteilen von Daten verdeutlicht. Ebene 1 zeigt die oberste Struktur mit dem zuletzt verwendeten Gruppenknoten G_0, von dem aus der Nachfolgeknoten bestimmt wird. Bei der Auswahl von G_1 wird in Ebene 2 gewechselt, da die Unterstruktur von G_1 wieder eine Gruppenstruktur ist. Hier wird nun von der Gruppe G_3 aus der Nachfolgeknoten bestimmt. Im Fall von G_4 wäre die Suche beendet, da dieser eine Voting-Struktur als Unterstruktur besitzt, aus der ein Quorum gebildet wird. Kann aus dieser Struktur jedoch kein Quorum gebildet werden, wird zurück in die Ebene 2 gewechselt und dort der nächste Knoten gesucht. In der Struktur ist nur noch der Gruppenknoten G_5 vorhanden. Hat G_5 auch eine Voting-Struktur aus der kein Quorum erstellt werden kann, muss zurück in Ebene 1 gewechselt werden, da alle Gruppen in Ebene 2 erfolglos benutzt wurden. Von Ebene 1 wird nun über die Gruppe G_2 weiter nach einem gültigen Quorum gesucht.

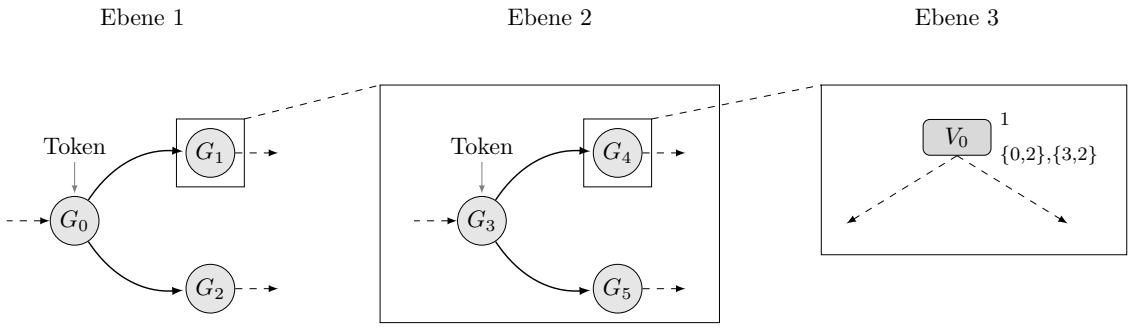


Abbildung 3.5: Gruppenstruktur

Sobald die Operation mit einem finalen Quorum erfolgreich durchgeführt wurde, werden die Token jeder verwendeten Gruppenstruktur auf die zuletzt benutzten Gruppen gesetzt. Dadurch werden sie zum Startpunkt einer erneuten Anfrage.

Beispiel 7 Um Energie zu sparen und die Replikate dennoch regelmäßig über das Netzwerk zu verteilen, könnten die Gruppen aus Abbildung 3.4 auf Seite 32 folgendermaßen benutzt werden:

- G_0, G_1 und G_2 enthalten Voting-Strukturen, die nahe liegende Knoten zum Sender benutzen. Dies entspricht einer geringen Flächenverteilung und geringen Energiekosten.

- G_3 und G_4 enthalten Voting-Strukturen, die Knoten mit mittlerer Entfernung zum Sender beinhalten, wodurch eine mittlere Flächenverteilung und mittlere Energiekosten entstehen.

- G_5 und G_6 enthalten Voting-Strukturen, die Knoten mit weiter Entfernung zum Sender benutzen. Durch die gegebene Struktur werden die Replikate einmal bei jedem Durchlauf auf einer weiten Fläche verteilt. Dies verursacht jedoch auch die höchsten Kosten.

Es ist bei einer Einteilung nach Energiekosten natürlich erforderlich, dass die Positionen der Sensorknoten und somit die Kosten zueinander bekannt sind.

3.5 Daten auslesen

Zum Auslesen der Daten müssen die Strukturen anders als beim Verteilen verwendet werden. Durch das Einsetzen der oben beschriebenen Rückverweise kann ein vollständiger View aus der zuletzt genutzten Voting-Struktur erstellt werden. Mit Hilfe des Views kann dann der aktuelle Objektzustand konstruiert werden. Somit muss innerhalb der Replikationsstrategie die zuletzt genutzte Voting-Struktur ermittelt werden.

Um das Auffinden dieser Struktur zu vereinfachen, wird in der Historie zusätzlich zu dem Timestamp, der Operation und den Daten auch die verwendete Gruppe und eine Versionsnummer im Replikat gespeichert. Die Versionsnummer muss eindeutig sein und kann durch eine Ganzzahl dargestellt werden, die bei jeder Operation um eins erhöht wird. Sie kann des Weiteren zur Komprimierung der Historie genutzt werden. Alle Einträge mit einer Versionsnummer kleiner als die von der Basisstation zuletzt gelesene Nummer könnten zum Beispiel gelöscht werden, da diese Daten schon verarbeitet wurden.

Durch die Rückverweise wird die partielle Historie der Knoten zur Erstellung des Views immer vollständig übertragen, wodurch eine erhöhte Kommunikation entsteht. Unter gewissen Umständen können die Rückverweise jedoch zur Energieeinsparung weggelassen werden. Dies ist zum Beispiel bei einer *enqueue* Operation der Fall, die einen Wert in einer Warteschlange einreiht. Da bei dem Einreihen keine Daten ausgelesen werden, benötigt die Operation keine Kenntnisse über den Zustand des Objekts. Das initiale Quorum der Operation ist somit leer und es werden keine Historieneinträge zur Erstellung des Objekts angefordert. Um das Objekt nun wieder aus der Strategie herzustellen, ist ein nativer Ansatz, alle in der Strategie vorkommenden Voting-Strukturen zur Erstellung eines Views zu nutzen. Dies birgt zwar einen erhöhten Aufwand beim Auslesen der Daten, jedoch wird der Aufwand zum Verteilen der Daten verringert.

Ein großer Nachteil bei dem Verzicht auf die Rückverweise ist die Anfälligkeit für Flächenschäden, da die Historieneinträge jeweils nur in einer Strukturen vorhanden sind und eventuell nur auf nahe beieinander liegenden Knoten verteilt wurden. Fallen zuviele Knoten dieser Strukturen aus, können keine Quoren mehr gebildet werden, um an diese Einträge zu gelangen und so das Objekt vollständig wiederherzustellen.

Kann auf die Sicherheit vor Flächenschäden nicht verzichtet werden, können Energiekosten eingespart werden, indem nur soviele Einträge der Historie übertragen werden wie maximal unbekannte Änderungen vorhergegangen sein können. Dadurch wird ein View erstellt, der alle vorhergegangenen Änderungen seit der Benutzung dieser Struktur enthält, die dann innerhalb dieser Struktur verteilt werden.

Das kostengünstige Auslesen der Daten durch die Gruppenstrukturen wird in der zu dieser Diplomarbeit parallel laufenden Bachelor-Arbeit [Peu10] von Christoph Peuser behandelt.

Kapitel 4

Implementierung des Rahmenwerkes

Dieses Kapitel beschreibt die Implementierung des Rahmenwerkes. Dafür wird zuerst ein Überblick über die Architektur und den verwendeten Komponenten gegeben. Die Implementierung basiert auf dem Betriebssystem *TinyOS* in der Version 2.x vom 16.07.2009 und ist auf Sensorknoten des Typs *MicaZ* ausgelegt. Nach dem allgemeinen Überblick über die Architektur werden die benutzten TinyOS Komponenten kurz vorgestellt. Darauf folgt eine genaue Beschreibung der für das Rahmenwerk entwickelten Komponenten, welche die Replikationsstrategien, die Replikate und den Programmablauf enthalten. Das Kapitel schließt mit den Designentscheidungen, möglichen Erweiterungen und Optimierungen, sowie Funktionstests ab.

4.1 Architektur

In Abbildung 4.1 auf der nächsten Seite ist die Gesamtarchitektur des Rahmenwerkes in Anlehnung an nesdoc, der Darstellungsform von Komponenten in TinyOS, dargestellt. Die grau hinterlegten Komponenten sind Eigenentwicklungen und die nicht hinterlegten Komponenten sind die benutzten Standardkomponenten von TinyOS. Das Kernstück des Rahmenwerkes stellt das ReplicationC Modul dar, da es den Programmablauf enthält. Die VotingStructureC Komponente implementiert das in Kapitel 3 vorgestellte Konzept der Gruppen- und Voting-Strukturen, während die ReplicationManagerC Komponente die Replikate speichert und die Operationen auf diesen anbietet.

Nachfolgend werden die verwendeten Standardkomponenten von TinyOS vorgestellt, bevor dann eine detaillierte Beschreibung der für das Rahmenwerk erstellten Komponenten erfolgt.

4.1.1 TinyOS Komponenten

Als TinyOS Komponenten werden in dieser Arbeit die Komponenten bezeichnet, die das Betriebssystem TinyOS bereitstellt. Diese Komponenten beinhalten in der Regel Grundfunktionen zum Ansteuern der Sensorknoten, wie zum Beispiel das Ein- und Ausschalten von Leuchtdioden. Für tiefer gehende Informationen zu den einzelnen Komponenten als die hier dargestellten sei auf [LG09] verwiesen.

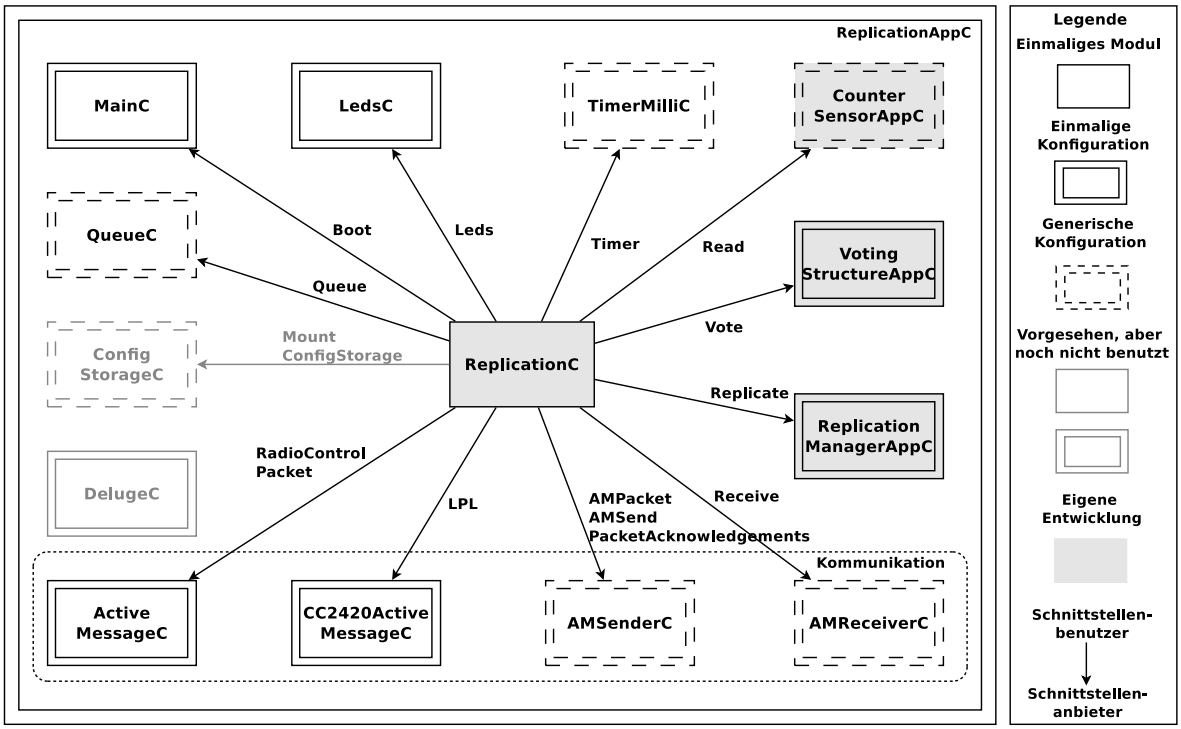


Abbildung 4.1: Komponenten der Gesamtarchitektur des Rahmenwerkes

MainC

Die MainC Komponente signalisiert durch das Ereignis *booted* über die *Boot* Schnittstelle, dass das Betriebssystem initialisiert und hochgefahren wurde. Komponenten, die vor dem *booted* Ereignis initialisiert werden müssen, sollten die Schnittstelle *SoftwareInit* der MainC benutzen, da diese nach der Hardware Initialisierung aber noch vor dem *booted* Ereignis aufgerufen wird.

Im Rahmenwerk dient die Komponente dazu, das Abtasten der Sensoren und das Verteilen der Daten nach dem Hochfahren des Knotens zu starten.

LedsC

Mit Hilfe der LedsC Komponente ist eine einfache Steuerung der Leuchtdioden von Sensorknoten möglich. Sie implementiert die Schnittstelle *Leds*, die unter anderem Kommandos zum Einschalten, Ausschalten und Wechseln des aktuellen Zustands der Leuchtdioden bietet. Die Leuchtdioden werden innerhalb des Rahmenwerkes dazu genutzt, Fehlverhalten sichtbar zu machen.

TimerMilliC

Die TimerMilliC Komponente stellt Timer bereit, die über die *Timer* Schnittstelle periodisch oder einmalig gestartet werden. Sobald ein Timer abgelaufen ist, löst dieser einen Event aus, auf den reagiert werden kann. Das Rahmenwerk setzt vier Timer ein, die folgende Aufgaben haben:

SampleTimer Dieser Timer wird periodisch gestartet und veranlasst das Abtasten des Sensors. Falls der Knoten nicht mit dem Replizieren von Daten beschäftigt ist, wird ein Replikationsvorgang nach dem Abtasten des Sensors eingeleitet.

CommunicationTimer Der CommunicationTimer dient als Zeitrahmen, in dem auf Antworten von anderen Sensorknoten gewartet wird. Kommen innerhalb dieses Zeitrahmens alle erwarteten Antworten an, löst der Timer aus und eine Fehlerbehandlung wird durchgeführt.

TimestampGenerator Der TimestampGenerator ist ein periodischer Timer, der den Timestamp für die Historien der Replikate bei jedem Auslösen um eins erhöht.

SendTimer Der SendTimer wird gestartet, sobald eine Nachricht versendet werden soll und der Sensorknoten noch eine andere Nachricht sendet. Die Anfrage zum Senden wird eine zeitlang zurückgestellt bis der Timer auslöst.

QueueC

Die QueueC Komponente stellt Warteschlangen mit fester Größe und festem Datentyp bereit, die bei der Initialisierung bestimmt werden. Die Schnittstelle *Queue* bietet Funktionen zum Einreihen und Entfernen eines Elements in eine Warteschlange, als auch das Auslesen der momentanen Anzahl von eingereihten Elementen. Im Rahmenwerk werden zwei Warteschlangen eingesetzt:

DataQueue In dieser Warteschlange werden die ausgelesenen Sensordaten mit ihrem Auslesezeitpunkt und einem Verweis auf eine Operation eingereiht, bevor sie unter Verwendung der Operation im Netzwerk repliziert werden.

SendQueue Die Sende-Warteschlange dient zum Einreihen von Nachrichten, die gesendet werden sollen. Dabei wird in jedem Element der Empfänger, der Typ der Nachricht, wie zum Beispiel eine Zusage und die dazugehörige Operation gespeichert.

Kommunikationskomponenten

Das Rahmenwerk enthält eine Single-Hop Kommunikation, die durch folgende Komponenten bereitgestellt wird:

ActiveMessageC Diese Komponente dient zum Starten und Stoppen der Radio-Einheit des Sensorknotens.

CC2420ActiveMessageC Diese Komponente stellt Funktionen speziell für einen CC2420 Radio-Chip bereit, der in Sensorknoten vom Typ MicaZ verbaut ist und gegen die das Rahmenwerk programmiert wurde. Durch sie kann im Rahmenwerk das sogenannte *Low Power Listening* eingesetzt werden. Beim *Low Power Listening* wird der Radio-Chip in regelmäßigen Abständen eingeschaltet und auf Nachrichten gewartet. Falls keine Nachrichten ankommen, wird der Chip bis zur nächsten Empfangsphase abgeschaltet, wodurch weniger Energiekosten durch die Empfangseinheit verursacht werden. Damit eine Nachricht mit Sicherheit empfangen wird, setzt der Sender je nach Protokoll beispielsweise vor die eigentliche Nachricht eine Präambel, die etwas länger als die Empfangsperiode der Empfangseinheit des Empfängers ist.

AMSenderC Die Funktionen zum Senden von Nachrichten über die Radio-Einheit wird von dieser Komponente bereitgestellt. Außerdem kann mit dieser Komponente eine Empfangsbestätigung einer Nachricht angefordert werden.

AMReceiverC Durch diese Komponente können Nachrichten über die Radio-Einheit empfangen werden.

ConfigStorageC

Die ConfigStorageC Komponente wird zum Speichern von Konfigurationsdaten benutzt und stellt dafür einen einfachen Zugriff auf den Flash-Speicher des Sensorknotens bereit. Konfigurationsdaten, die gespeichert werden, sind zum Beispiel die Abtastrate des Sensors oder die Periode des Timestamp Generators. Für den Zugriff auf den Flash-Speicher muss für diesen ein Layout angegeben werden, das ihn durch die Angabe von Startpunkten und Speicherlängen in verschiedene Bereiche teilt. Dabei sind die minimalen Speichereinheiten zu beachten. Ein MicaZ Knoten besitzt zum Beispiel einen Flash-Speicher vom Typ At45db und dessen kleinste Einheit beträgt 256B[1].

Die Komponente wurde im Rahmenwerk für den späteren Einsatz auf realen Knoten vorgesehen, um neue Konfigurationen im Netzwerk zu speichern, die auch nach einem Neustart der Sensorknoten beibehalten werden. Da die Komponente von dem im nächsten Kapitel beschriebenen Simulator *TOSSIM* nicht unterstützt wird, wurde sie zwar im Rahmenwerk vorgesehen, aber nicht in die Simulation mit eingebunden und getestet.

DelugeC

Durch den Einsatz der DelugeC Komponente können Anwendungen auf den Sensorknoten leicht ausgetauscht werden. Dafür speichert die Komponente die verschiedenen Anwendungen auf dem Flash-Speicher der Sensorknoten und startet diese bei Bedarf von dort. Weiter können die einzelnen Anwendungen auch durch andere Anwendungen überschrieben werden, die über das Netzwerk an alle Sensorknoten verteilt werden. Somit ist ein leichter Austausch von verschiedenen Strategien möglich, indem die verschiedenen Strategien in das Rahmenwerk integriert und dieses dann auf alle Knoten im Netzwerk verteilt wird, ohne dabei jeden Knoten einzeln programmieren zu müssen.

Wie die ConfigStorageC Komponente ist der Einsatz der DelugeC Komponente lediglich im Rahmenwerk für einen späteren Einsatz auf realen Sensorknoten vorgesehen. Die korrekte Funktionsweise konnte aufgrund fehlender Unterstützung des Simulators *TOSSIM* nicht getestet werden und wird somit in die Simulation nicht mit eingebunden.

4.1.2 CounterSensorAppC Komponente

Die CounterSensorAppC Komponente besitzt die Schnittstelle *Read* und ist ein Dummy. Sie dient als Ersatz für reale Sensoren und erzeugt bei jedem Auslesen des Sensors einen künstlichen Wert. Der erste Wert beginnt bei Null und wird bei jeder erneuten Anfrage um eins erhöht bis

1 http://www.tinyos.net/tinyos-2.x/doc/html/tep103.html (18.11.2010)

zu $2^{16} - 1$. Danach findet ein Überlauf der Variablen statt und die Zählung fängt wieder bei null an.

Die Komponente wird eingesetzt, um die Korrektheit der Replikation aufzuzeigen. Ein korrekt repliziertes Objekt muss die ausgelesenen Sensorwerte in richtiger und vollständiger Reihenfolge von null bis zum zuletzt generierten Wert aufweisen.

4.1.3 VotingStructureAppC Komponente

Die VotingStructureAppC Komponente enthält die Replikationsstrategien, die zum Verteilen der Daten im Netzwerk genutzt werden und stellt die Funktionalität zum Bilden der Quoren aus den Strategien bereit. Die Abbildung 4.2 zeigt die Konfiguration der Komponente. Die RandomC Komponente generiert Zufallswerte, die dazu dienen, eine Kante aus einer Menge von Kanten mit gleicher Priorität auszuwählen. Mit der MainC Komponente wird das VotingStructureC Modul während des Startens des Knotens initialisiert.

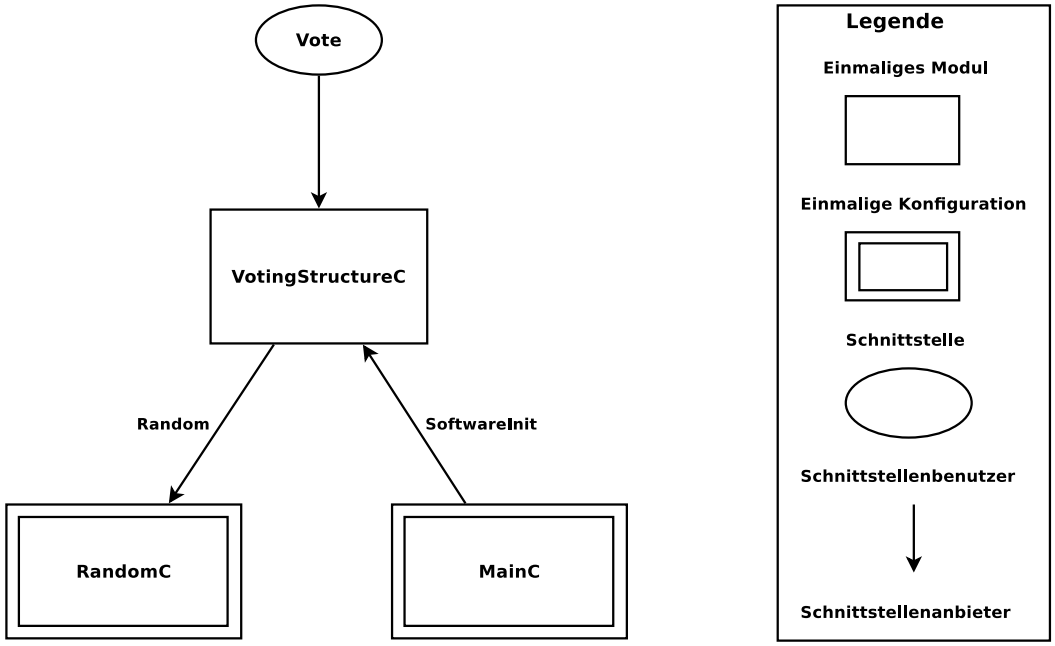


Abbildung 4.2: VotingStructureAppC Konfiguration

Die VotingStructureAppC Komponente stellt die Schnittstelle *Vote* bereit, die im Quelltext 4.1 auf der nächsten Seite zu sehen ist. Über die Schnittstelle kann nach einem Quorum gefragt werden und die Token, nach einer erfolgreichen Benutzung eines finalen Quorums, auf die zuletzt verwendeten Gruppen der Gruppenstruktur umgesetzt werden. Dadurch wird das nächste Quorum ausgehend von diesem Knoten gebildet.

Die Anfrage nach einem Quorum ist in ein Kommando und ein Ereignis aufgeteilt und wird durch das Kommando *getQuorum* initialisiert. Dem Kommando wird eine Quorenstruktur übergeben, auf der das Quorum aufgebaut wird. Sobald die Strategie nach einem Quorum durchsucht wurde, wird dies über das Ereignis *getQuorumDone* bekannt gegeben. Konnte ein

```
1 Interface Vote
2 {
3   command void getQuorum(Quorum_t * quorum);
4   event   void getQuorumDone(error_t error);
5   command void setNextToken(Quorum_t * quorum);
6 }
```

Quelltext 4.1: Schnittstelle Vote

gültiges Quorum gebildet werden, gibt *getQuorumDone* den Erfolg bekannt und das Quorum befindet sich in der übergebenen Struktur und kann benutzt werden. Konnte kein gültiges Quorum gebildet werden, wird ein Misserfolg über *getQuorumDone* signalisiert und in der übergebenen Struktur befindet sich ein ungültiges Quorum, dass nicht zur Replikation eingesetzt werden darf.

Die Quorenstruktur die *getQuorum* übergeben wird, ist in Abbildung 4.3 dargestellt. Sie speichert die Identifikationsnummer der Gruppe aus der Gruppenstruktur, aus dessen Voting-Struktur das Quorum gebildet wurde. Außerdem enthält die Struktur die Funktion, für die das Quorum gebildet werden soll. Diese muss schon vor der Übergabe festgelegt werden. Das eigentliche Quorum besteht aus einem Array, das die Identifikationsnummer der Knoten speichert. Weiter wird in der Quorenstruktur festgehalten, wie die Knoten auf die Anfrage der auszuführenden Operation geantwortet haben. Dabei werden die Antworten gezählt und erst ausgewertet, wenn alle Knoten des Quorums geantwortet haben. Die genaue Vorgehensweise zur Benutzung des Quorums werden in Abschnitt 4.1.5 auf Seite 53 beschrieben.

Neben den Quorenstrukturen sind in der Abbildung 4.3 auch die Datenstrukturen der Gruppenstrukturen und der Votingstrukturen dargestellt. Die zuletzt verwendete Gruppe der obersten Gruppenstruktur wird durch einen Token markiert, um von ihr aus die nächste Gruppe zum Erstellen eines Quorums zu bestimmen. Eine Gruppe enthält eine Identifikationsnummer und Kanten, die die Nachfolgegruppen und deren Priorität beinhalten. Weiter enthält jede Gruppe Informationen über die Struktur, die sich unter ihr befindet. Ist die Unterstruktur eine Gruppenstruktur, zeigt die Gruppe auf die zuletzt verwendete Untergruppe. Bei einer Voting-Struktur wird immer auf den Wurzelknoten der Struktur verwiesen.

Eine Votingstruktur besteht aus virtuellen und physikalischen Knoten, Kanten und Rückverweisen. Die virtuellen Knoten haben eine Identifikationsnummer, eine Stimmenanzahl, die Mindeststimmenanzahlen der Operationen und Kanten. Die Kanten speichern die Identifikationsnummer des Zielknotens und ob es sich bei dem Zielknoten um einen virtuellen oder physikalischen Knoten handelt. Zusätzlich enthält jede Kante eine Priorität. Da physikalische Knoten nur als Blattknoten vorkommen, enthalten diese nur eine Identifikationsnummer, eine Stimmenanzahl und einen Status, der angibt ob sie ihre Stimme abgeben können oder nicht. Der Status wird nach jeder fehlgeschlagenen Quorumsanfrage aktualisiert. Dabei werden die Knoten, die mit einem *Abort* die Anfrage beantwortet haben, als beschäftigt eingetragen und somit nicht erneut bei der Bildung eines Quorums berücksichtigt, bis sie zurückgesetzt werden.

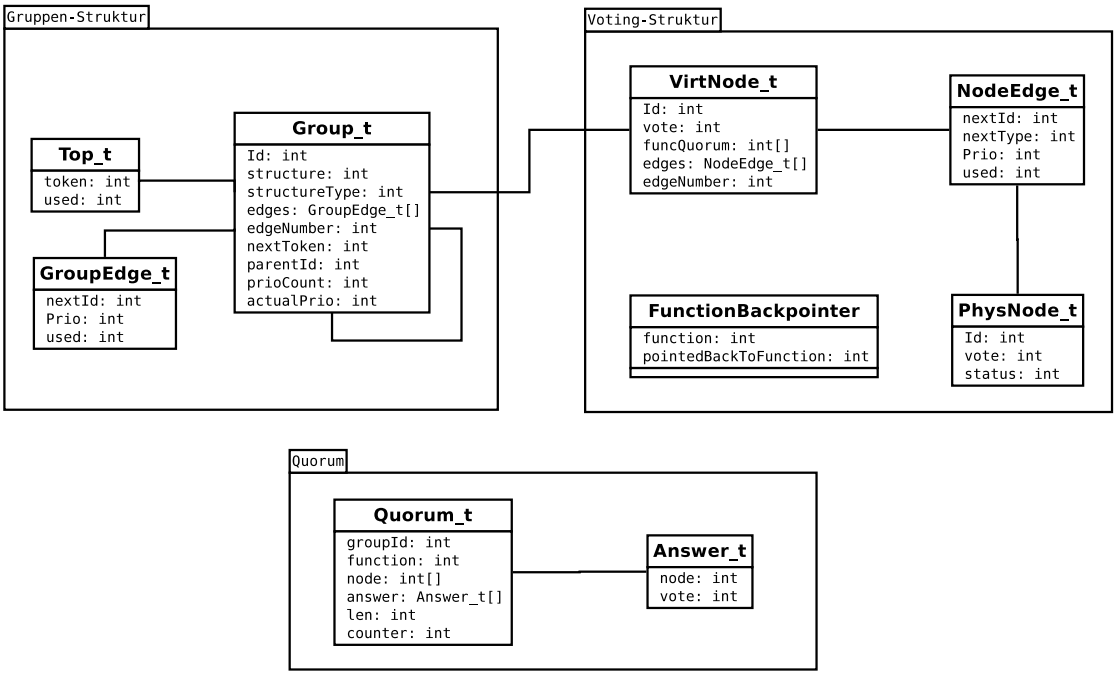


Abbildung 4.3: VotingStructureAppC Datenstrukturen

Beispiel einer Replikationsstrategie

Nun wird an einem Beispiel erklärt, wie eine Replikationsstrategie innerhalb der Komponente definiert ist, bevor auf den Ablauf zur Bildung eines Quorums eingegangen wird. Die Abbildung 4.4 auf der nächsten Seite zeigt die grafische Darstellung der Replikationsstrategie, die hier exemplarisch auf die Datenstrukturen abgebildet wird. Die Gruppen G_0 und G_2 enthalten als Unterstrukturen Voting-Strukturen. Die Gruppe G_1 besitzt dagegen als Unterstruktur erneut eine Gruppenstruktur, dessen Gruppen dann erst auf Voting-Strukturen verweisen. Die Voting-Strukturen gehen von zwei Operationen aus, die nachfolgend als *enqueue* und *dequeue* Operationen bezeichnet werden.

Im Quelltext 4.2 wird zuerst die Reihenfolge der Mindeststimmenanzahl der Operationen bestimmt. Der erste Eintrag ist hier die Mindeststimmenanzahl für ein initiales Quorum der *enqueue* Operation. Darauf folgt dann die zur erfolgreichen Ausführung erforderliche Stimmenanzahl für ein finales Quorum der *enqueue* Operation. Als Nächstes kommt die Stimmenanzahl für ein initiales Quorum der *dequeue* Operation und zuletzt die Stimmenanzahl für ein finales Quorum der *dequeue* Operation. Diese Reihenfolge muss nun bei jedem Knoten in den Voting-Strukturen eingehalten werden.

```
1 //Function Order in Votingstructure
2 enum Functiontype
3 {
4   ENQUEUEINIT = 0, ENQUEUEFINAL, DEQUEUEINIT, DEQUEUEFINAL
5 };
```

Quelltext 4.2: Beispiel Replikationsstrategie: Reihenfolge der Mindeststimmenanzahl

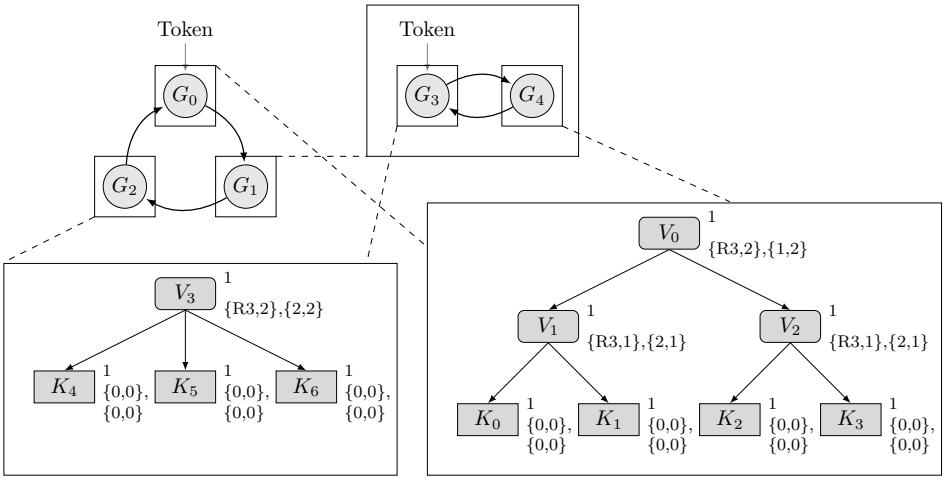


Abbildung 4.4: Beispiel Replikationsstrategie

Zunächst werden die Gruppenstrukturen im Quelltext 4.3 definiert. Dafür wird zuerst ein Array mit der Größe der Anzahl der Gruppen in der Strategie erstellt. In diesem Array wird jede Gruppe eingetragen. Es ist zu beachten, dass die Gruppen von Null angefangen gezählt werden und es keine Lücken innerhalb der Nummerierung gibt. Die Gruppe G_0 ist an der nullten Stelle des Arrays. Die Gruppe G_1 an der ersten Stelle und so weiter. Ein Gruppeneintrag beinhaltet die Identifikationsnummer, die Informationen der Unterstruktur und die zugehörigen Kanten. Danach müssen noch Standardwerte gesetzt werden, die als Hilfswerte dienen und bei der Initialisierung der Knoten immer gleich sind.

```
// GROUPS
Group_t groups[5] =
  //changeable:  id, structure, structureType, edges, #edges
  //default values: nextToken, parentId, prioCount, actualPrio
    { 0, 0, VIRT,  group0Edges, 1, -1, -1, 0, -1},
    { 1, 3, GROUP, group1Edges, 1, -1, -1, 0, -1},
    { 2, 3, VIRT,  group2Edges, 1, -1, -1, 0, -1},
    { 3, 3, VIRT,  group3Edges, 1, -1, -1, 0, -1},
    { 4, 0, VIRT,  group4Edges, 1, -1, -1, 0, -1},
};

// GROUP EDGES
// G0 -> G1 Prio 0 (not used(default value))
GroupEdge_t group0Edges[1] = {{1,0,-1}};
// G1 -> G2 Prio 0 (not used(default value))
GroupEdge_t group1Edges[1] = {{2,0,-1}};
GroupEdge_t group2Edges[1] = {{0,0,-1}};
GroupEdge_t group3Edges[1] = {{4,0,-1}};
GroupEdge_t group4Edges[1] = {{3,0,-1}};
```

Quelltext 4.3: Beispiel Replikationsstrategie: Gruppenstrukturen

Nach den Gruppen werden die Kanten der Gruppen beschrieben. Dafür wird für jede Gruppe ein Array erstellt, das die Kanteninformationen beinhaltet. Die Arrays sind jeweils so groß wie die Kantenanzahl, die von den Gruppen abgeht. In diesem Beispiel enthält jede Gruppe nur eine Kante, wodurch jedes Array nur ein Element enthält. Für jede Kante wird die Identifikationsnummer der Nachfolgegruppe gespeichert und ihre Priorität angegeben. Zusätzlich muss ein Hilfswert angegeben werden, der anzeigt, ob die Kante benutzt wurde. Dieser Wert ist bei der Initialisierung immer derselbe. Das Gruppenarray und die Kantenarrays beschreiben zusammen alle Gruppenstrukturen der Replikationsstrategie.

Die Voting-Strukturen des Beispiels werden im Quelltext 4.4 erstellt. Dafür werden zunächst die virtuellen Knoten definiert, die wie die Gruppen von null angefangen und ohne Lücken durchnummeriert werden. Der nullte Knoten befindet sich an der nullten Stelle des Arrays, der erste Knoten an der ersten Stelle und so weiter. Neben der Identifikationsnummer und der Stimme, die sie abgeben können, werden die Mindeststimmenanzahlen für die Operationen in der Reihenfolge, wie im Quelltext 4.2 auf Seite 43 vorgeschrieben, angegeben. Außerdem werden die noch zu definierenden Arrays der Kanten und die Kantenanzahl den einzelnen Knoten zugewiesen.

```
1  // VIRT NODES
2  VirtNode_t virtNodes[4] =
3  {  //id, vote, {func1.init, func2.final, ...}, edges, edgeNumber
4     { 0 , 1    , {0         , 2     ,1,2}, virtNode0Edges , 2},
5     { 1 , 1    , {0         , 1     ,2,1}, virtNode1Edges , 2},
6     { 2 , 1    , {0         , 1     ,2,1}, virtNode2Edges , 2},
7     { 3 , 1    , {0         , 2     ,2,2}, virtNode3Edges , 3}
8  };
9  // PHYS NODES
10 PhysNode_t* physNodes;
11 PhysNode_t physNodes1[6] = //{id, vote, status}
12   {{1,1,FREE}, {2,1,FREE}, {3,1,FREE}, {4,1,FREE}, {5,1,FREE}, ↩
       ⟶{6,1,FREE}};
13 PhysNode_t physNodesDefault[6] =
14   {{5,1,FREE}, {6,1,FREE}, {2,1,FREE}, {1,1,FREE}, {3,1,FREE}, ↩
       ⟶{4,1,FREE}};
15
16 // VIRT NODE EDGES
17 NodeEdge_t virtNode0Edges[2] = //id,type,prio,used
18     {{1,VIRT,0,-1}, {2,VIRT,0,-1}};
19 NodeEdge_t virtNode1Edges[2] =
20     {{0,PHYS,0,-1}, {1,PHYS,0,-1}};
21 NodeEdge_t virtNode2Edges[2] =
22     {{2,PHYS,0,-1}, {3,PHYS,0,-1}};
23 NodeEdge_t virtNode3Edges[3] =
24     {{4,PHYS,0,-1}, {5,PHYS,0,-1}, {6,PHYS,0,-1}};
25
26 // Function Back Pointer
27 int numberOfBackPointer = 1;
28 FunctionBackPointer_t functionBackPointer[1] = {{ENQUEUEINIT, ↩
     ⟶DEQUEUEINIT}};
```

Quelltext 4.4: Beispiel Replikationsstrategie: Voting-Strukturen

Im Quelltext 4.4 auf der vorherigen Seite folgt nach den virtuellen Knoten die Definition der physikalischen Knoten. Dafür werden so viele Arrays in der Größe der Anzahl der physikalischen Knoten angelegt, wie verschiedene Zuordnungen von realen Sensorknoten auf die physikalischen Knoten der Replikationsstrategie nötig sind. Das ist erforderlich, da jeder reale Sensorknoten unterschiedliche Entfernungen zu den einzelnen realen Knoten besitzt. Wenn die Strategie so ausgelegt ist, dass hauptsächlich Knoten in der Nähe des Senders benutzt werden, muss für jeden Knoten bestimmt werden, welche Knoten nahe bei ihm liegen. In diesem Beispiel gibt es eine Zuordnung für einen realen Knoten mit der Identifikationsnummer eins und eine Zuordnung für die restlichen Knoten. In der praktischen Anwendung hat jeder reale Knoten eine eigene Zuordnung.

Der physikalische Knoten K_0 befindet sich an der Stelle Null des Arrays, der Knoten K_1 ist an Stelle eins und so weiter. Die realen Knoten werden nun durch ihre Identifikationsnummer zugewiesen. Dem realen Sensorknoten mit der Nummer eins ist für K_1 der Replikationsstrategie der reale Knoten mit der Nummer zwei zugeordnet. Den anderen realen Sensorknoten ist für K_1 der reale Sensorknoten mit der Nummer sechs zugeordnet. Somit benutzt jeder reale Sensorknoten die gleiche Replikationsstrategie, verwendet aber unterschiedliche Repräsentationen der physikalischen Knoten.

Nun müssen die virtuellen und physikalischen Knoten durch Kanten miteinander verbunden werden. Dafür erhält jeder virtuelle Knoten ein Array in der Größe seiner abgehenden Kanten. Jede Kante im Array speichert den Typ des Zielknotens und dessen Identifikationsnummer. Weiter wird die Priorität der Kante festgehalten. Wie bei den Kanten der Gruppen, muss zusätzlich ein Hilfswert angegeben werden, der für jede Kante bei der Initialisierung der Gleiche ist.

Zuletzt werden die Rückverweise der Voting-Strukturen bestimmt. In diesem Fall verweist der initiale Aufruf der *enqueue* Operation auf den initialen Aufruf der *dequeue* Operation. Das bedeutet, dass das initiale Quorum der *enqueue* Operation aus einem initialen Quorum der *dequeue* Operation der zuletzt verwendeten Voting-Struktur gebildet wird.

Bei der Initialisierung der VotingStructureAppC Komponente muss abhängig vom realen Sensorknoten die Gruppe bestimmt werden, die in der obersten Ebene der Gruppenstrukturen durch den Token markiert ist. Zusätzlich muss noch das Array mit den physikalischen Knoten dem realen Sensorknoten zugeordnet werden. Dies ist im Quelltext 4.5 zu sehen und durch eine *Switch-Case* Anweisung realisiert.

Mit den vorgestellten Definitionen wurde die Replikationsstrategie aus Abbildung 4.4 auf Seite 44 als Datenstrukturen dargestellt, aus denen die Quoren in der Komponente gebildet werden.

Ablauf der Quorenbildung

Nach der Überführung der Replikationsstrategien in die Datenstrukturen für die VotingStructureAppC Komponente, wird nun der Ablauf der Quorenbildung auf diesen Strukturen beschrieben. Durch das Schnittstellenkommando *getQuorum* wird das in Abbildung 4.5 auf Seite 48 dargestellte Aktivitätsdiagramm gestartet.

```
1  command error_t Init.init()
2  {
3    switch(TOS_NODE_ID) //Select Node
4    {
5      case 1:
6        top.token = 0;
7        top.used = -1; //default Value
8        physNodes = physNodes1;
9        break;
10     default:
11       top.token = 0;
12       top.used = -1; //default Value
13       physNodes = physNodesDefault;
14   }
15   ...
16 }
```

Quelltext 4.5: Beispiel Replikationsstrategie: Initialisierung

Zuerst wird unterschieden, ob für die Operation ein Rückverweis angegeben ist. Ist dies der Fall, wird die zuletzt verwendete Gruppe der obersten Ebene der Gruppenstrukturen als Suchgruppe eingesetzt. Falls die Unterstruktur dieser Suchgruppe wieder eine Gruppe ist, wird diese als neue Suchgruppe eingesetzt, da sie auf der Ebene zuletzt verwendet wurde. Dies wird solange wiederholt, bis die Unterstruktur eine Voting-Struktur ist, aus der dann ein Quorum gebildet wird. In diesem Quorum befindet sich bei einer korrekten, nicht probabilistischen Voting-Struktur mindestens ein Knoten der letzten Operation.

Ist kein Rückverweis für die Operation angegeben, wird bei dem ersten Aufruf nach dem Umsetzen der Token eine Gruppe nach den Kantenprioritäten, ausgehend von der zuletzt verwendeten Gruppe, ausgewählt. Diese berechnete Gruppe enthält als Unterstruktur eine Voting-Struktur aus der dann ein Quorum gebildet wird.

Konnte das Quorum nicht erfolgreich zur Verteilung von Daten genutzt werden, muss erneut *getQuorum* aufgerufen werden. Bei diesem zweiten Durchgang wird wieder versucht, ein Quorum aus der Voting-Struktur der letzten Suchgruppe zu erstellen. Schlägt das fehl, muss eine neue Suchgruppe bestimmt werden, die als Unterstruktur eine Voting-Struktur besitzt.

Die Abbildung 4.6 auf Seite 49 zeigt den Ablauf zur Bestimmung der Suchgruppen. Zuerst wird überprüft, ob noch unbenutzte Kanten bei der aktuellen Priorität vorhanden sind. Sind noch Kanten unbenutzt, wird zufällig eine dieser Kanten ausgewählt und die Zielgruppe der Kante ist die neue Suchgruppe. Befindet sich als Unterstruktur dieser Gruppe eine Voting-Struktur, wird diese Gruppe zurückgeben. Ist die Unterstruktur jedoch eine Gruppe, wird diese als neue Suchgruppe bestimmt und überprüft, ob noch unbenutzte Kanten in der Priorität vorhanden sind. Trifft dies nicht zu, wird die nächste Priorität der Kanten angenommen. Sind Kanten mit der neuen Priorität vorhanden, wird zufällig eine dieser Kanten ausgewählt. Sind keine Kanten in der neuen Priorität vorhanden, wurden alle Kanten, die von der Gruppe abgehen erfolglos benutzt. Aus diesem Grund darf es keine Lücken innerhalb der Prioritätenvergabe der Kanten geben.

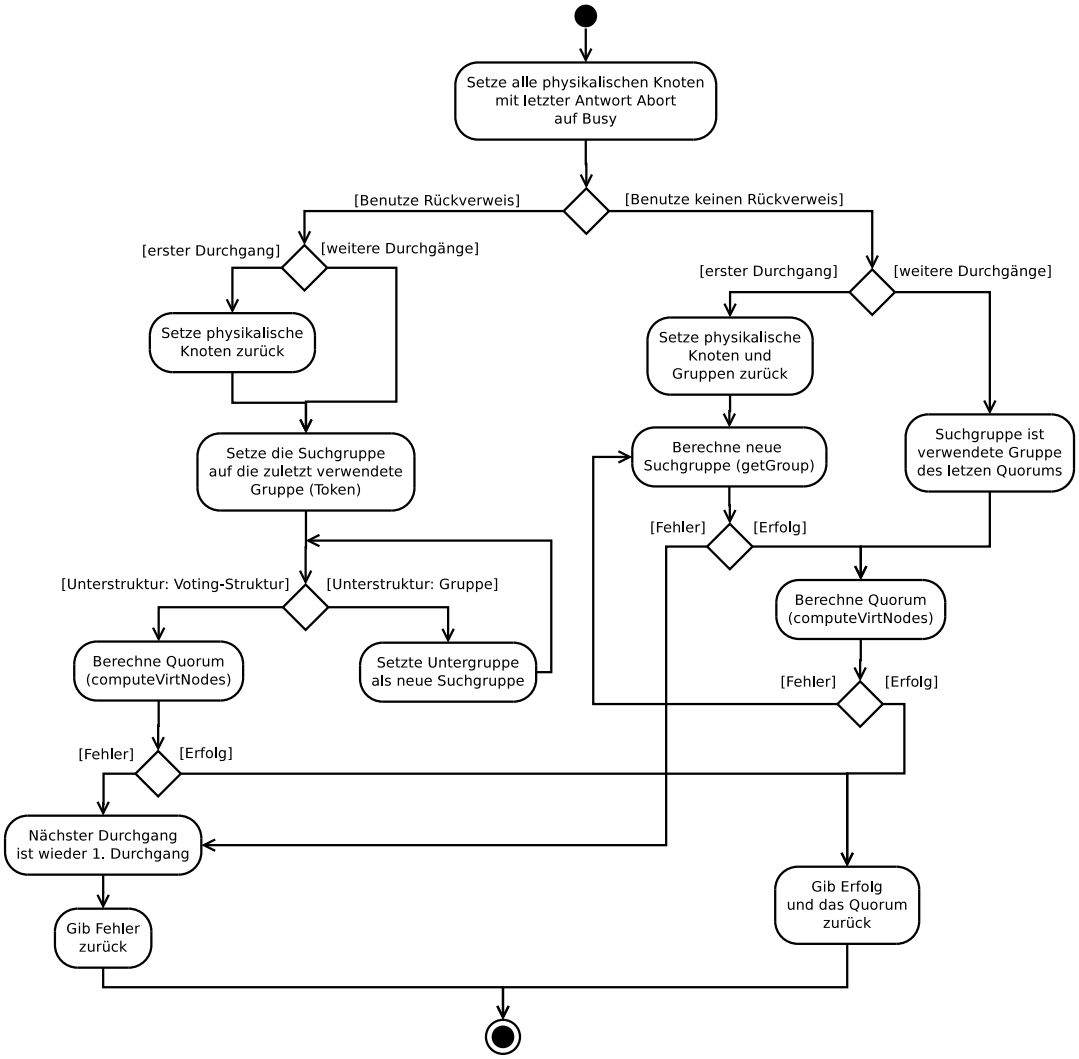


Abbildung 4.5: Ablauf zur Berechnung eines Quorums

Sind in der obersten Ebene der Gruppenstrukturen alle Kanten benutzt worden, wird ein Fehler zurückgegeben, da keine Gruppe zur Benutzung bestimmt werden konnte. Sind alle Kanten der Suchgruppe einer Unterebene benutzt worden, wird in eine Ebene höher gewechselt, um dort von der zuletzt verwendeten Gruppe die nächste Gruppe zu bestimmen.

Nachdem eine Gruppe mit einer Voting-Struktur gefunden wurde, wird ein Quorum mit Hilfe dieser Struktur gebildet. Der Ablauf dafür ist rekursiv und ist in Abbildung 4.7 auf Seite 50 zu sehen. Zuerst wird zwischen einem virtuellen und einem physikalischen Knoten unterschieden. Wird der Ablauf mit einem virtuellen Knoten aufgerufen, wird überprüft, ob dieser genug Mindeststimmen seiner Kindknoten für die auszuführende Operation zusammen hat. Ist dies der Fall, gibt der Knoten seine Stimme ab. Ist dies nicht der Fall, wird überprüft, ob noch Kanten mit der aktuellen Priorität vorhanden sind. Falls keine Kanten vorhanden sind, muss die Priorität um eins erhöht werden. Sind daraufhin immer noch keine Kanten verfügbar, verweigert

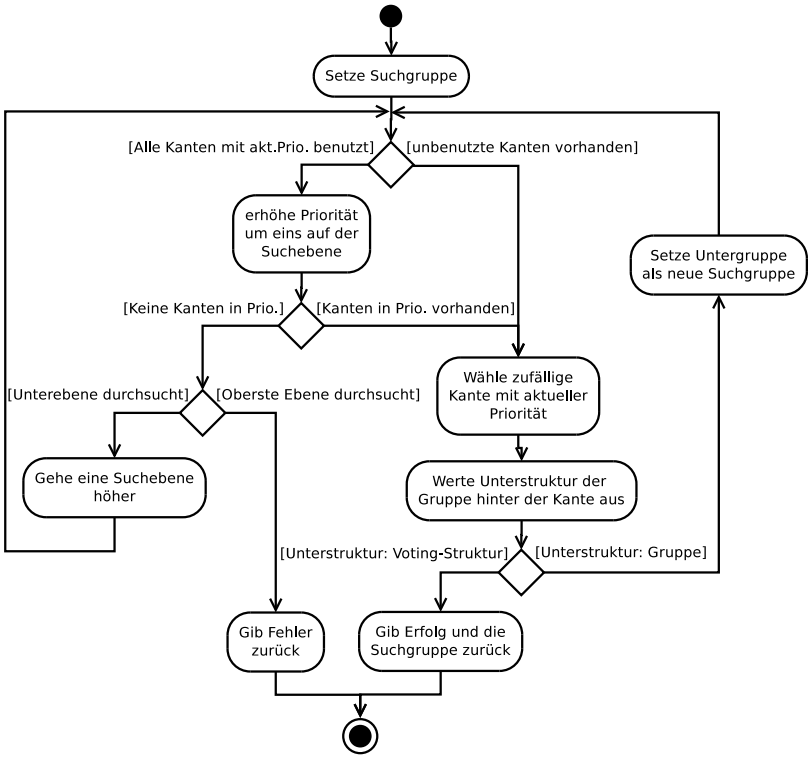


Abbildung 4.6: Ablauf zur Berechnung einer Suchgruppe auf einer Gruppenstruktur

der Knoten seine Zustimmung. Sind jedoch noch Kanten unbenutzt, wird zufällig eine dieser Kanten mit der aktuellen Priorität ausgesucht und der Ablauf mit dem dahinter liegenden Knoten erneut aufgerufen.

Bei einem physikalischen Knoten wird sein Status überprüft. Ist der Knoten beschäftigt, verweigert er seine Zustimmung. Ist er frei, wird er zu der Menge des Quorums hinzugefügt und gibt seine Stimme an seinen Vaterknoten ab. Nachdem die Stimme abgegeben wurde, durchläuft der Vaterknoten wieder eine Überprüfung der Mindeststimmen.

Sobald der Wurzelknoten genug Stimmen gesammelt hat, gibt er seine Stimme ab. Sind alle Kanten des Wurzelknotens benutzt worden, ohne das die Mindeststimmenanzahl der Operation erreicht wurde, verweigert der Wurzelknoten seine Zustimmung. Wenn der Wurzelknoten seine Stimme abgibt oder verweigert, wird der Ablauf endgültig verlassen und der aufrufende Ablauf aus Abbildung 4.5 weitergeführt.

Nach der erfolgreichen Benutzung eines finalen Quorums müssen die Token, die die zuletzt verwendeten Gruppen markieren, umgesetzt werden. Das ist in Abbildung 4.8 auf der nächsten Seite dargestellt. Dazu wird zuerst die verwendete Gruppe ermittelt, die als Unterstruktur die Voting-Struktur beinhaltet, aus der das Quorum gebildet wurde. Diese Gruppe wurde innerhalb der Quoren-Datenstruktur festgehalten und ist somit direkt bestimmbar. Sie wird als die Suchgruppe eingesetzt. Nun wird unterschieden, ob die Suchgruppe sich in der obersten Ebene der Gruppenstrukturen befindet. Ist die Suchgruppe in einer Unterstruktur, muss der

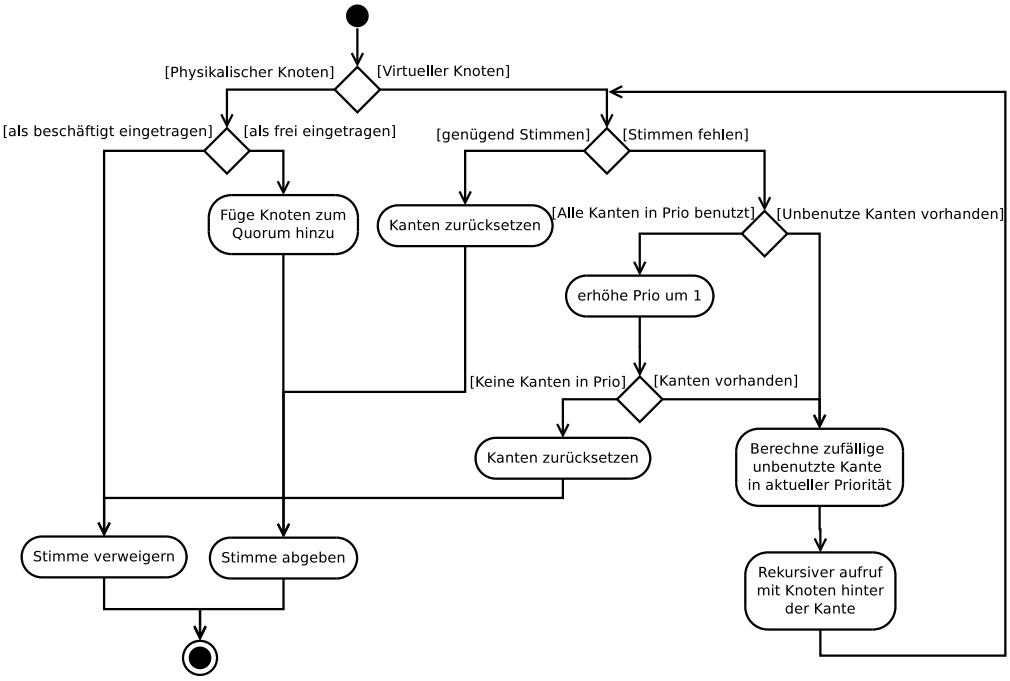


Abbildung 4.7: Ablauf der Bildung eines Quorums auf einer Voting-Struktur

Token der höheren Ebene in Form des Verweises auf die Unterstruktur umgesetzt werden. Die Gruppe der höheren Ebene, die als Unterstruktur auf den Vaterknoten der Suchgruppe verweist, wird so umgeändert, dass die Unterstruktur nun die Suchgruppe ist. Danach wird diese Gruppe als Suchgruppe eingesetzt. Sobald die Suchgruppe sich in der obersten Ebene der Gruppenstrukturen befindet, wird der oberste Token auf die Suchgruppe umgesetzt. Dadurch zeigt nun jede verwendete Gruppe jeweils auf die verwendete Gruppe der darunter liegenden Ebene.

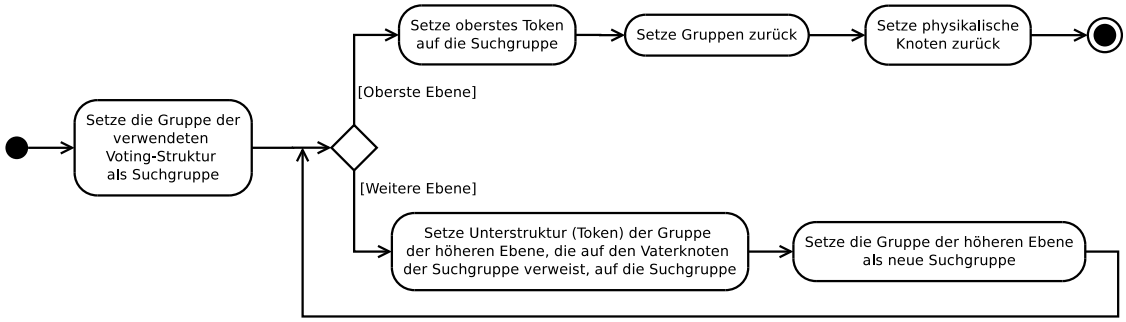


Abbildung 4.8: Umsetzen der Token nach erfolgreicher Nutzung eines Quorums

Beispiel 8 Gegeben sei die Struktur in Abbildung 4.4 auf Seite 44. Die zuletzt verwendete Gruppe in der obersten Ebene ist die Gruppe G_0 und von dieser wird die nächste Gruppe gesucht. Dies ist die Gruppe G_1, die als Unterstruktur wieder eine Gruppenstruktur besitzt, indem ihre Unterstruktur auf den Gruppenknoten G_3 verweist. Von G_3 aus wird nun die nächste

Gruppe gesucht. In diesem Fall ist das G_4, welche eine Voting-Struktur als Unterstruktur besitzt. Werden nach einer erfolgreichen Benutzung eines Quorums aus der Voting-Struktur die Token umgesetzt, muss die Unterstruktur der Gruppe G_1 von G_3 auf G_4 umgesetzt werden. Danach wird auf eine Ebene höher gewechselt und der oberste Token auf G_1 gesetzt. Die nächste Anfrage nach einem Quorum beginnt dann von G_1 aus.

4.1.4 ReplicationManagerAppC Komponente

Die ReplicationManagerAppC Komponente speichert die Replikate und stellt die Operationen auf diesen bereit. Abbildung 4.9 zeigt die Konfiguration der Komponente. Das ReplicationManagerC Modul enthält dabei die eigentliche Implementierung und die MainC Komponente dient zum Initialisieren des Moduls während der Startphase der Sensorknoten.

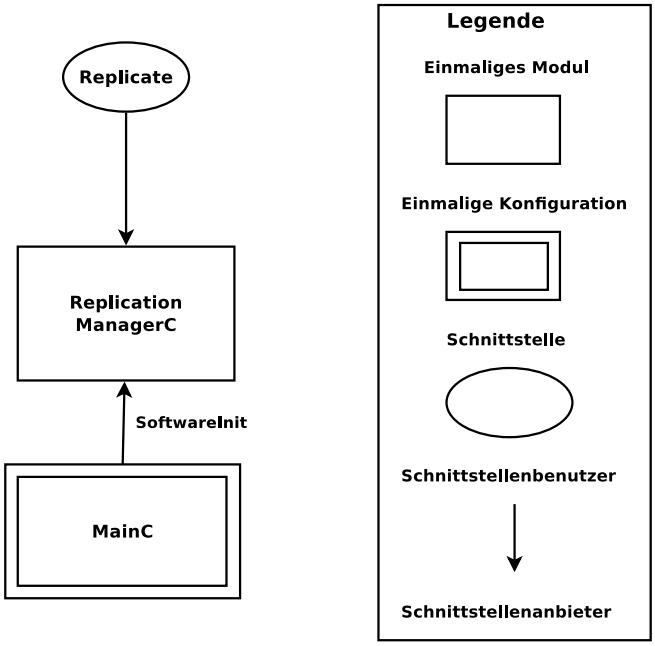


Abbildung 4.9: ReplicationManagerAppC Konfiguration

Die Komponente bietet die Schnittstelle *Replicate* an, die im Quelltext 4.6 auf der nächsten Seite dargestellt ist. Da die Darstellung der Replikate und der Operationen auf den Replikaten durch das General Quorum Consensus-Verfahren frei gestaltbar ist, gibt der Quelltext nur ein Beispiel wieder, das in diesem Rahmenwerk genutzt wird. Auch die in Abbildung 4.10 auf der nächsten Seite dargestellte Datenstruktur der Replikate ist nur eine Möglichkeit, die hier vorgestellt wird. Die Schnittstelle und die Datenstruktur sind somit nicht fest vorgegeben und sollten für den Anwendungszweck angepasst werden.

Die Datenstruktur der Replikate enthält in diesem Rahmenwerk eine Identifikationsnummer, eine Versionsnummer, die bei jeder neuen Verteilung des Replikats um eins erhöht wird und eine Gruppenidentifikationsnummer, die angibt, welche Gruppe aus der Replikationsstrategie für das Verteilen des Replikats genutzt wurde. Des Weiteren enthalten die Replikate Informationen darüber, wie viele Operationen gleichzeitig lesend, beziehungsweise schreibend auf dem Replikat

```
1  Interface Replicate
2  {
3     // ENQUEUE INIT
4    command error_t enqueueInit(uint8_t id, Replicate_t * repli);
5    command error_t enqueueInitCommit(uint8_t);
6    command error_t enqueueInitAbort(uint8_t);
7
8     // ENQUEUE FINAL
9    command error_t enqueueFinal(uint8_t id, Replicate_t newHistory);
10   command error_t enqueueFinalCommit(uint8_t id);
11   command error_t enqueueFinalAbort(uint8_t id);
12
13   ...
14 }
```

Quelltext 4.6: Schnittstelle Replicate

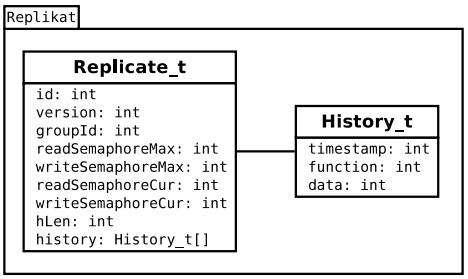


Abbildung 4.10: Beispiel-Datenstruktur der Replikate

durchgeführt werden dürfen. Weiter besitzt das Replikat eine Historie, die die auf dem Replikat ausgeführten Operationen mit ihren Datenwerten und ihren Ausführungszeitpunkten speichert.

Die Operationen zum Einreihen der Daten in die Warteschlange, nachfolgend *enqueue* Operation genannt, sind nach dem General Quorum Consensus-Verfahren in eine initiale und finale Phase eingeteilt, die im Rahmenwerk jeweils noch einmal in Aufruf und Abschluss der Operation unterteilt werden. Die *enqueue* Operation wird bei einem initialen Quorum mit der Funktion *EnqueueInit* aufgerufen und mit *EnqueueInitCommit* bei einer erfolgreichen Ausführung und mit *EnqueInitAbort* bei einem Misserfolg der Ausführung abgeschlossen. Mit einem finalen Quorum wird zuerst die Funktion *EnqueueFinal* aufgerufen und mit *EnqueueFinalCommit* im Erfolgsfall und mit *EnqueueFinalAbort* bei einem Misserfolg beendet. Die Funktionen verhalten sich dabei folgendermaßen:

EnqueueInit Es wird versucht eine Leseberechtigung zu bekommen. Ist dies erfolgreich, wird das vollständige Replikat zurückgegeben und der Erfolg gemeldet. Konnte keine Leseberechtigung erworben werden, wird der Misserfolg gemeldet.

EnqueueInitCommit Die Leseberechtigung wird wieder abgegeben.

EnqueueInitAbort Die Leseberechtigung wird wieder zurückgegeben.

EnqueueFinal Es wird versucht eine Schreibberechtigung zu bekommen. Konnte die Berechtigung nicht erworben werden, wird ein Misserfolg gemeldet. Ansonsten wird eine an die Funktion übergebene Historie mit der Historie des Replikats vereinigt und als eine temporäre Historie gespeichert. Zu Testzwecken wird derzeit die Historie verworfen und nur der neuste Eintrag der beiden Historien gespeichert, falls sich mehr als 50 Einträge in der Historie befinden. Das simuliert das Auslesen und nachfolgende Löschen der Historie.

EnqueueFinalCommit Das Replikat wird durch das temporär gespeicherte Replikat ersetzt. Danach wird die Schreibberechtigung zurückgegeben.

EnqueueFinalAbort Das temporäre Replikat wird verworfen und die Schreibberechtigung zurückgegeben.

4.1.5 ReplicationC Komponente

Die ReplicationC Komponente enthält den eigentlichen Programmablauf und ist wie in Abbildung 4.1 auf Seite 38 zu sehen, mit allen vorgestellten Komponenten verbunden. Sobald das Betriebssystem hochgefahren ist, erhält die Komponente ein Signal und beginnt daraufhin mit der Initialisierung des Programmablaufs. Dieser ist in Abbildung 4.11 dargestellt. Dafür werden die benötigten Variablen mit Anfangswerten belegt und die Radio-Einheit gestartet. Danach werden der *TimestampGenerator Timer* und der *Sample Timer* der TimerMilliC Komponente gestartet.

Abbildung 4.11: ReplicationC: Bootphase

Durch das Auslösen des *Sample Timers* beginnt der eigentliche Programmablauf aus Abbildung 4.12 auf der nächsten Seite. Nach dem Auslösen des Timers wird der Sensor ausgelesen und die ausgelesenen Daten mit dem Auslesezeitpunkt in einer Warteschlange eingereiht. Falls der Knoten zu diesem Zeitpunkt noch Daten repliziert, wird kein weiterer Repliziervorgang eingeleitet. Ist der Knoten jedoch nicht beschäftigt, beginnt das Verteilen der gerade ausgelesenen Sensordaten durch die Verwendung des *Zwei-Phasen-Commit Protokolls* aus Abschnitt 2.2.1 auf Seite 14.

Dafür wird ein initiales Quorum zum Verteilen der Daten über die VotingStructureAppC Komponente erstellt und jedem Knoten des Quorums eine Anfrage zum Replizieren der Daten gesendet. Gleichzeitig wird der *CommunicationTimer* gestartet, der die Zeitspanne bestimmt, in der die Antworten der befragten Knoten eintreffen müssen. Sind in dieser Zeitspanne nicht alle Antworten eingetroffen, werden allen Knoten, die der Anfrage zugestimmt haben und allen Knoten die nicht geantwortet haben, ein globaler Abbruch gesendet. Den Knoten ohne Antwort wird der Abbruch gesendet, da deren Nachricht eventuell nicht empfangen worden ist, diese aber der Ausführung zugestimmt haben. Falls diese Knoten ausgefallen sind und dadurch nicht antworten können, entsteht an dieser Stelle ein blockierendes Verhalten des *Zwei-Phasen-Commit Protokolls*. Da dies den kompletten Ablauf stoppt, sollte dieses Verhalten für einen realen Einsatz optimiert werden. Ansätze dafür werden in Abschnitt 4.3 auf Seite 57

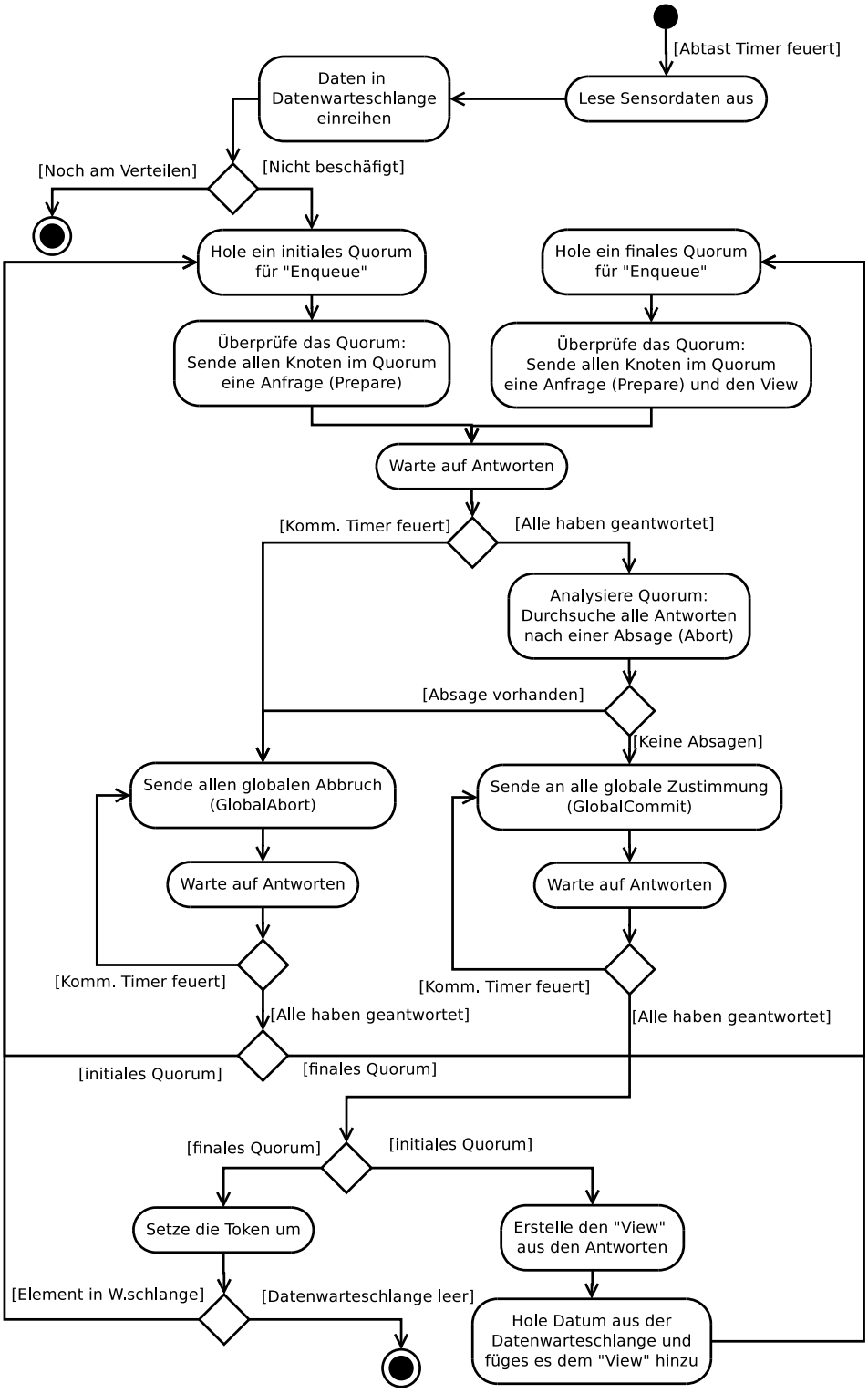


Abbildung 4.12: ReplicationC: Programmablauf der Replikation

genannt. Nachdem alle Bestätigungen des Abbruchs eingetroffen sind, wird ein neues Quorum angefordert und das Versenden der Anfragen wird von vorne begonnen.

Treffen alle Antworten auf die Anfrage zum Replizieren der Daten innerhalb der vorgegebenen Zeitspanne ein, werden diese analysiert. Sollte sich eine Absage unter den Antworten befinden, wird allen zustimmenden Knoten ein globaler Abbruch gesendet und ein neues Quorum angefordert. Auch hier besteht wieder die Gefahr einer Blockierung, da auf alle Antworten gewartet werden muss. Bei der Anforderung eines neuen Quorums werden die Knoten mit einer Absage als beschäftigt in die Voting-Struktur der VotingStructureAppC Komponente eingetragen, so dass sie zur Erstellung des neuen Quorums ihre Stimme verweigern.

Haben alle gefragten Sensorknoten der Anfrage zugestimmt, wird ihnen eine globale Zustimmung zum Abschließen der Operation zugesendet und erneut auf ihre Bestätigung gewartet. Nachdem alle geantwortet haben, werden die Historien, die mit der Zustimmung mitgesendet wurden, zu einem *View* vereint und das Datum des Sensorknotens aus der Warteschlange an den View mit Timestamp und der Art der Operation angehängt. Dieser View muss nun an ein finales Quorum gesendet werden, um die Daten zu replizieren. Dafür wird ein finales Quorum über die VotingStructureAppC Komponente angefordert. Den Knoten des Quorums wird dann unter Verwendung des *Zwei-Phasen-Commit Protokolls* eine Anfrage zusammen mit dem View gesendet. Sobald dies erfolgreich abgeschlossen werden konnte, müssen die Token der VotingStructureAppC Komponente umgesetzt werden. Befinden sich nun noch Daten in der Warteschlange, wird ein neuer Zyklus zum Replizieren der Daten eingeleitet. Ansonsten stoppt der Programmfluss bis zum nächsten Auslösen des *Sample Timers*.

Die Abbildung 4.13 auf der nächsten Seite zeigt das Verhalten der Knoten eines initialen Quorums, die eine Anfrage zum Einreihen der zu replizierenden Daten bekommen. Zuerst rufen diese die einleitende Funktion der angefragten Operation der ReplicationManagerAppC Komponente auf. Gibt die Funktion einen Fehler zurück, wird dem anfragenden Knoten eine Absage gesendet. Ist die Anfrage dieser Operation jedoch erfolgreich, wird die Zustimmung und die von der Funktion zurückgegebene Historie an den anfragenden Knoten gesendet. Danach wartet der Knoten auf eine Antwort, ob die Operation endgültig abgeschlossen werden soll oder abgebrochen werden muss. Je nach Antwort werden die entsprechenden Funktionen der Operation ausgeführt und eine Bestätigung zurückgesendet. Für einen Operationsaufruf mit einem finalen Quorum ist der Ablauf der Gleiche, nur dass dabei die entsprechenden *Final* Funktionen der ReplicationManagerAppC Komponente aufgerufen werden und das es nicht nötig ist, die Historie bei der Zusage mitzuschicken.

Nachrichten, die das Rahmenwerk durch Sicherheitsfunktionen mehrmals versendet, werden vom Empfänger verworfen, falls dieser die Nachricht schon einmal empfangen hat. Dies wird durch einen Vergleich der Timestamps der Nachricht realisiert. Die wiederholt versendeten Nachrichten bekommen den Timestamp der ersten Nachricht mitgesendet. Sobald der Empfänger die Nachricht erhält, speichert er den Timestamp und kann diesen mit weiteren ankommenden Nachrichten vergleichen.

Die Anfrage zum Replizieren von Daten kann parallel zum Verteilen der eigenen Sensordaten ausgeführt werden. Wenn ein Knoten also den Programmablauf aus Abbildung 4.12 auf der vorherigen Seite durchläuft, kann er Anfragen anderer Knoten beantworten und der Replikation dieser Daten zustimmen. Des Weiteren kann ein Sensorknoten gleichzeitig der Replikation

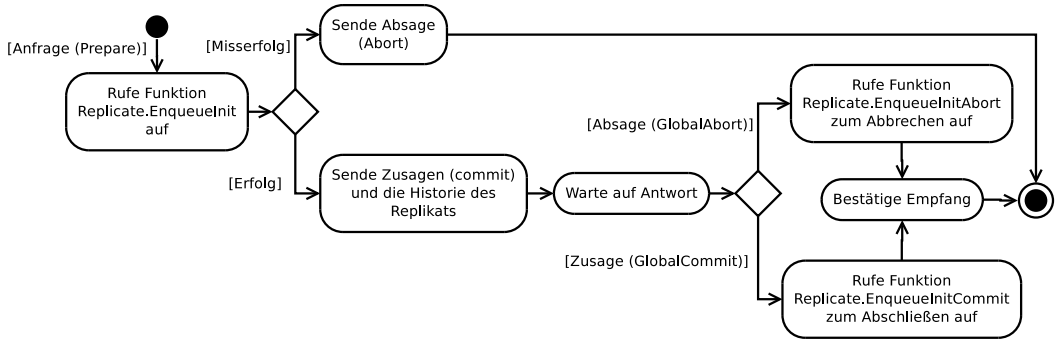


Abbildung 4.13: ReplicationC: Verarbeitung einer Anfrage

mehrerer Replikate zustimmen. Wenn also zwei Knoten gleichzeitig ihre Daten auf einem Knoten replizieren möchten, kann dieser Knoten beiden die Ausführung bestätigen. Nur der gleichzeitige Zugriff auf ein Replikat wird verweigert. Dieser Fall tritt zum Beispiel ein, wenn eine Basisstation ein Replikat auf einem Sensorknoten ausliest, während ein anderer Sensorknoten versucht, das Replikat auf diesem Knoten zu ändern.

4.2 Designentscheidungen

Für die Realisierung des Rahmenwerkes wurden drei grundlegende Designentscheidungen getroffen, die nachfolgend beschrieben werden. Zuerst wird auf die Kapselung der Kommunikation in der ReplicationC Komponente eingegangen. Danach wird die Entscheidung einer Single-Hop Kommunikation und die Verwendung des Zwei-Phasen-Commit Protokolls erläutert.

4.2.1 Keine Kommunikation in VotingStructureAppC

Wird ein Quorum von der VotingStructureAppC Komponente angefordert, geht diese zunächst davon aus, dass alle Blattknoten, die die realen Sensorknoten repräsentieren, der auszuführenden Operation zustimmen. Erst danach werden die Sensorknoten des Quorums befragt, ob sie der Operation wirklich zustimmen. Alle Knoten, die ihre Zustimmung verweigern, werden dann als beschäftigt in die Voting-Struktur eingetragen, so dass sie bei der nun erforderlichen neuen Bildung eines Quorums ihre Stimme verweigern. Die Voting-Struktur muss somit mehrmals durchlaufen werden, bis ein Quorum gefunden wurde, dessen Knoten alle der Operation zustimmen.

Damit der Baum nur einmal durchlaufen wird, könnten die Blattknoten direkt nach ihrer Zustimmung befragt werden. Dadurch würde ein Quorum von der VotingStructureAppC Komponente zurückgegeben werden, dessen Knoten vollständig der Operation zugestimmt haben.

Diese Variante wurde jedoch nicht gewählt, da die Kommunikation und dadurch auch das Kommunikationsprotokoll auf die zwei Komponenten VotingStructureAppC und ReplicationC

aufgeteilt wäre. Damit wäre auch der Programmablauf geteilt, wodurch sich die Komplexität des Rahmenwerkes erhöhen würde.

4.2.2 Single Hop-Kommunikation

Im Rahmenwerk war zunächst eine Multi Hop-Kommunikation mit der TinyOS Komponente *Tymo* vorgesehen, die jedoch nicht korrekt durch den verwendeten Simulator *TOSSIM* unterstützt wurde. Dies machte sich durch ein Übermaß an nicht versendbaren Nachrichten bemerkbar.

Neben diesem schwerwiegenden Aspekt bietet die *Tymo* Komponente keine Schnittstelle um festzustellen, ob die gesendete Nachricht beim Empfänger angekommen ist und müsste selbst implementiert werden. Aufgrund der Komplexität dieser Bestätigung, der fehlerhaften Simulation und dem zeitlichen Rahmen der Arbeit wurde vorerst auf eine Single Hop-Kommunikation zurückgegriffen.

4.2.3 Zwei-Phasen-Commit-Protokoll

In Sensornetzwerken spielt die Kommunikation zwischen den Sensorknoten eine der wichtigsten Rollen und besitzt dadurch ein großes Maß an Implementierungsaufwand. Um die Komplexität des Rahmenwerkes gering zu halten und einen funktionierenden Prototypen zu entwickeln, wurde das Zwei-Phasen-Commit Protokoll gewählt. Das Protokoll ist zwar blockierend, aber einfach strukturiert, wodurch die Grundfunktionalität des Rahmenwerkes besser evaluiert werden kann.

4.3 Erweiterungs- und Optimierungsmöglichkeiten

Speicherverbrauch optimieren

Aufgrund des prototypischen Designs des Rahmenwerkes wird übermäßig viel Speicher verbraucht. Um das Rahmenwerk auf realen Sensorknoten verwenden zu können, muss der Speicherverbrauch optimiert werden. Der größte Verbrauch entsteht durch die Datenhaltung der Replikate. Da davon ausgegangen wird, dass jeder Knoten im Netzwerk seine Daten repliziert, hält die ReplicationManagerAppC Komponente für das Replikat jedes Knotens Speicher bereit.

Wenn sichergestellt werden kann, dass ein Knoten niemals ein Replikat eines anderen Sensorknotens speichern muss, ist es nicht nötig den Speicher zu reservieren, wodurch sich der Verbrauch verringert. Da alle Replikate derzeit im sehr begrenzten RAM liegen, könnten diese in den Flash-Speicher der Sensorknoten ausgelagert werden, aus dem nur soviele Informationen wie nötig wieder in den RAM geladen werden, um weniger Energie durch das Kopieren der Daten zu verbrauchen. Das Speichern der Daten im Flash würde eine erhebliche Entlastung des RAMs ergeben.

Aber nicht nur in der ReplicationManagerAppC Komponente wird Speicher für die Replikate bereit gehalten, sondern auch in der ReplicationC Komponente, um die beiden Komponenten klar zu trennen. Da an den Knoten gleichzeitig mehrere Anfragen nach verschiedenen Replikaten gestellt werden können, wird für jedes Replikat Speicher zum Senden des Replikats freigehalten. Neben dem Speicher zum Senden, ist Speicher für jedes mögliche ankommende Replikat reserviert. Außerdem ist neben dem selbst erstellten View auch noch zusätzlich Speicher für jeden möglichen ankommenden View anderer Sensorknoten reserviert.

In dieser Komponente wird somit viel Speicher vorgehalten, der nicht unbedingt gleichzeitig genutzt wird. Dadurch ergibt sich hier ein hohes Optimierungspotenzial. Eine Möglichkeit besteht darin, einen Speicherpool zu generieren und aus diesem Pool Speicher für die gerade benötigte Aktion zu holen. Wenn der Speicherpool leer ist, kann die Anfrage nicht durchgeführt werden und ein Knoten verweigert beispielsweise seine Zustimmung zu einer Operation. Weiter wäre es möglich, nicht den maximal vorgesehenen Speicher eines Replikats zu reservieren, sondern nur einen Teil. Wenn also ein Replikat auf maximal 50 Historieneinträge beschränkt wird, aber derzeit nur zehn Einträge besitzt, muss auch nur Speicher für zehn Einträge alloziert sein.

Neben den Replikaten könnte zusätzlicher Speicher bei Voting-Strukturen der Replikationsstrategie gespart werden, indem Teilbäume in den verschiedenen Strukturen wiederverwendet werden.

Struktur Parser

Zur einfacheren Umsetzung von Replikationsstrategien könnten diese, wie in [Sto06] beschrieben, in Anlehnung an der *DOT*-Beschreibungssprache des Programms GraphViz[1] beschrieben werden. Dies führt zu einer einfacheren Beschreibung der Strategie, da sie von Datenstrukturen gelöst ist. Diese Beschreibung könnte dann durch einen Parser eingelesen werden, der die Strategie auf die Datenstrukturen der VotingStructureAppC Komponente überträgt und ausgibt.

Synchronisation

Damit die Sensorwerte der einzelnen Sensorknoten miteinander in eine zeitliche Verbindung gebracht werden können, ist eine Synchronisation der Timestamp Generatoren der Knoten nötig. Zur Synchronisation könnten die Generatoren zum Beispiel durch eine Nachricht der Basisstation gestartet werden und durch diese regelmäßig neu synchronisiert werden. Weiter könnten auch die einzelnen Sensorknoten regelmäßig eine Synchronisation veranlassen. Dabei ist zu beachten, dass durch ein Zurücksetzen des Timestamps Operationen einen früheren Timestamp bekommen können, als Operationen, die vor ihnen ausgeführt wurden. Diese Problematik muss bei der Synchronisation sichergestellt werden.

Multi Hop-Kommunikation

Die Single Hop-Kommunikation des Rahmenwerkes sollte durch eine Multi Hop-Kommunikation erweitert werden, da in realen Sensornetzwerken eine direkte Kommunikation mit allen Knoten nicht gewährleistet ist.

1 http://www.graphviz.org/ (22.11.2010)

Bootroutine erweitern

Da nach dem Neustart eines Sensorknotens alle Daten im RAM verloren gehen, sind auch die ursprünglichen Markierungen der Gruppenstrukturen nicht mehr vorhanden. Um zumindest die Information der zuletzt verwendeten Gruppe wiederzubeschaffen, kann die Bootroutine angepasst werden. Dazu sollte nach dem Schema der Basisstation das letzte Replikat ermittelt werden, da dieses die Information der zuletzt verwendeten Gruppe beinhaltet.

Automatische Knotenzuweisung

Die Zuweisung der Sensorknoten zu den Blattknoten in der Replikationsstrategie erfolgt im Rahmenwerk vor der Auslegung der Knoten. Somit müssen die Positionen der einzelnen Knoten zueinander von vornherein bekannt sein. Sollen die Sensorknoten jedoch in einem schwer zugänglichen Terrain ausgelegt werden und müssen sie dafür zum Beispiel aus einem Flugzeug ausgeworfen werden, sind die Positionen der Knoten zueinander nicht vorhersehbar.

Dieses Problem könnte durch eine automatische Zuweisung nach dem Auslegen erfolgen, in dem die Knoten über die Radio-Einheit ihre Nachbarknoten und deren Entfernungen ermitteln und den Blattknoten der Voting-Strukturen dann erst zugewiesen werden. Die Entfernung könnte dabei durch Laufzeitmessungen der Nachrichten oder durch schrittweises Erhöhen der Sendereichweite erfolgen.

4.4 Funktionstests

Die nächsten Abschnitte beschreiben, wie die selbstentwickelten Komponenten des Rahmenwerkes auf Fehler getestet wurden. Die Tests sind mit dem in Abschnitt 5.1 auf Seite 63 beschriebenen Simulator *TOSSIM* durchgeführt worden.

CounterSensorAppC Komponente

Die CounterSensorAppC Komponente gibt Werte aufsteigend von Null bis zu $2^{16} - 1$ zurück. Um die Funktion zu testen, wurde der Sensor wiederholt ausgelesen und der Rückgabewert überprüft, der immer um eins höher als der Vorgängerwert sein muss.

VotingStructureAppC Komponente

Die VotingStructureAppC Komponente gibt Debugnachrichten über jeden ausgewählten Knoten und jeder ausgewählten Gruppe zurück. Die Ausgaben wurden mit den verwendeten Strukturen abgeglichen. Dafür wurden von vornherein physikalische Knoten als beschäftigt eingetragen und Kanten als benutzt markiert. Diese Kanten und Knoten durften somit bei dem Auswahlverfahren nicht verwendet werden.

Im Quelltext 4.7 auf der nächsten Seite ist eine Ausgabe der VotingStructureAppC Komponente mit dem Simulator *TOSSIM* dargestellt. In der Ausgabe wird von der Gruppe Null aus gesucht und die Gruppe Eins ausgewählt. Diese enthält eine Voting-Struktur als Unterstruktur (SType: 1), die mit dem Knoten Null beginnt (Structure: 0) und von dem aus ein Quorum gebildet wird. Der Knoten besitzt fünf Kanten (numEdges: 5), die alle die Priorität Null besitzen

(PrioCount: Prio 0; Count: 5). Außerdem benötigt der Knoten drei Stimmen seiner Kindknoten (needed Votes: 3) für die erfolgreiche Ausführung der Operation. Die erste ungenutzte Kante, die in dieser Ausgabe ausgewählt wird, ist die fünfte Kante, hinter der sich der physikalische Knoten mit der Identifikationsnummer sechs befindet. Am Schluss wird das gebildete Quorum noch einmal zusammenhängend ausgegeben.

```
1  DEBUG (1): startGroup ID: 0
2  DEBUG (1): GroupVote SUCCESS
3  DEBUG (1): startGroup NextToken: 1
4  DEBUG (1): nextGroup ID: 1
5  DEBUG (1): nextGroup Structure: 0
6  DEBUG (1): nextGroup SType: 1
7  DEBUG (1): nextGroup ParentID: 0
8
9  DEBUG (1): VirtNode ID: 0
10 DEBUG (1): VirtNode numEdges: 5
11 DEBUG (1): VirtNode needed Votes: 3
12 DEBUG (1): VirtNode PrioCount: Prio: 0; Count: 5
13 DEBUG (1): VirtNode randomEdge: 5
14 DEBUG (1): VirtNode edge nextID: 6
15 DEBUG (1): VirtNode edge nextType: 0
16 DEBUG (1): PhysNode ID: 7
17 DEBUG (1): PhysNode Vote: 1
18 DEBUG (1): VirtNode randomEdge: 3
19 DEBUG (1): VirtNode edge nextID: 2
20 DEBUG (1): VirtNode edge nextType: 0
21 ...
22 DEBUG (1): EndVote: 1
23
24 DEBUG (1): Quorum GroupId: 1
25 DEBUG (1): Quorum Len: 3
26 DEBUG (1): Quorum Counter: 0
27 DEBUG (1): Quorum Node: 7
28 DEBUG (1): Quorum Node: 3
29 DEBUG (1): Quorum Node: 2
```

Quelltext 4.7: Funktionstest VotingStructureAppC

Kommunikationsablauf

Um die Kommunikation zwischen den Komponenten zu überprüfen, werden beim Empfangen und Absenden von Nachrichten Debugnachrichten ausgegeben. Mit diesen Informationen kann das Senden und Empfangen von Nachrichten und der korrekte Ablauf der Kommunikation überprüft werden. Ein korrekter Ablauf im Simulator *TOSSIM* ist im Quelltext 4.8 abgebildet.

```
1 DEBUG (1): Sending to 7 PREPARE
2 DEBUG (1): Need to send 1 history entrys to 7
3 DEBUG (7): I have received a message from 1 with type PREPARE ↩
    ⟶(with History Information)
4 DEBUG (1): Successfully sent.
5 DEBUG (1): Sending to 7 Prepare DATA. Package 1 of 1
```

```
6 DEBUG (7): I have received a message from 1 with type prepare ↩
    ⟶DATA. Packet 1 of 1 with hTimestamp 59
7 DEBUG (7): Sending to 1 COMMIT
8 DEBUG (1): Successfully sent.
9 DEBUG (1): I have received a message from 7 with type COMMIT
10 DEBUG (7): Successfully sent.
11 DEBUG (1): Sending to 7 GLOBALCOMMIT
12 DEBUG (7): I have received a message from 1 with type GLOBALCOMMIT
13 DEBUG (7): Sending to 1 ACKCOMMIT
14 DEBUG (1): Successfully sent.
15 DEBUG (1): I have received a message from 7 with type ACKCOMMIT
16 DEBUG (7): Successfully sent.
```

Quelltext 4.8: Funktionstest Kommunikationsablauf

In Versionen von TinyOS nach dem 16.07.2009 funktioniert laut [Sch10] (Seite 100) das Low Power Listening nicht korrekt auf realen Sensorknoten. Deswegen wird die Version vom 16.07.2009 genutzt. Da TOSSIM Low Power Listening nicht unterstützt, wurde es mit dem *AvroraZ*[1] Simulator getestet. Dieser simulierte in der Version 1.6 von TinyOS mitgelieferte Beispiele jedoch fehlerhaft, die auf realen Sensorknoten einwandfrei funktionierten. Aus diesem Grund wird das Low Power Listening im weiteren Verlauf deaktiviert.

ReplicationManagerAppC Komponente

Sobald die ReplicationManagerAppC Komponente die partielle Historie eines Replikats mit einem empfangenen View zusammenfügt, gibt sie die entstandene und gespeicherte Historie, wie im Quelltext 4.9 zu sehen, im Simulator aus.

```
1 DEBUG (3): ********** REPLICATE ID 1 of NODE 3 **************
2 DEBUG (3): *    Version:        3                            *
3 DEBUG (3): *    GroupId:        0                            *
4 DEBUG (3): *    Hlen   :        3                            *
5 DEBUG (3): *                                                 *
6 DEBUG (3): *    Entry 0 Timestamp:      59                   *
7 DEBUG (3): *    Entry 0 Function :      0                    *
8 DEBUG (3): *    Entry 0 Data     :      0                    *
9 DEBUG (3): *                                                 *
10 DEBUG (3): *    Entry 1 Timestamp:      119                  *
11 DEBUG (3): *    Entry 1 Function :      0                    *
12 DEBUG (3): *    Entry 1 Data     :      1                    *
13 DEBUG (3): *                                                 *
14 DEBUG (3): *    Entry 2 Timestamp:      179                  *
15 DEBUG (3): *    Entry 2 Function :      0                    *
16 DEBUG (3): *    Entry 2 Data     :      2                    *
17 DEBUG (3): *************************************************
```

Quelltext 4.9: Funktionstest ReplicationManagerC

1 http://citavroraz.sourceforge.net/ (22.11.2010)

Kapitel 5

Simulation und Ergebnisse

In diesem Kapitel werden die Simulation und die daraus resultierenden Ergebnisse beschrieben. Dafür wird zuerst der Simulator vorgestellt. Da dieser jedoch nicht alle Voraussetzungen zur Beweisführung liefert, wird in dem darauffolgenden Abschnitt ein Testmodell aufgestellt, das zusammen mit dem Simulator zur Erstellung der Ergebnisse genutzt wird. In Abschnitt 5.3 werden vier Testszenarien festgelegt, die verschiedene Replikationsstrategien beinhalten. Für diese Tests werden in Abschnitt 5.4 Annahmen zu ihrem Energieverbrauch getroffen. Diese Annahmen werden am Ende des Kapitels überprüft, indem die Ergebnisse miteinander verglichen werden. Dabei wird festgestellt, ob das Konzept der Gruppenstrukturen für Replikationsstrategien und die dadurch gezielte Verteilung von Replikaten, im Gegensatz zu klassischen Verfahren, Energie einspart.

5.1 Der Simulator

Für die Simulation des Rahmenwerkes wird der Simulator *TOSSIM* eingesetzt. *TOSSIM* ist der Standardsimulator von TinyOS, der zusammen mit dem Quellcode von TinyOS mitgeliefert wird. Er bietet Mechanismen zur Interaktion mit dem Netzwerk, wie das Einspeisen von Nachrichten. Des Weiteren enthält der Simulator eine Möglichkeit für Debugausgaben, denen Kategorien zugeordnet werden und die so leicht getrennt voneinander ein- und ausgeschaltet werden können. Die Erweiterung *TOSSIMz* des Simulators ermöglicht die Simulation des Energieverbrauchs der Sensorknoten. Da die Erweiterung jedoch das Low Power Listening noch nicht unterstützt, verbrauchen die Radio-Chips während der Simulation permanent Energie, egal ob sie eine Nachricht versenden oder empfangen oder ob sie untätig sind. Deswegen wurde als Ersatz der Simulator *AvroraZ* getestet. emphAvroraZ ist eine Erweiterung des Simulators *Avrora*, der für die Simulation von Atmel AVR Mikrocontroller und Mica2 Sensorknoten entwickelt wurde. Durch die Erweiterung AvroraZ werden zusätzlich MicaZ Sensorknoten unterstützt.

AvroraZ funktionierte in verschiedenen Tests leider nicht korrekt mit dem Low Power Listening von TinyOS. So wurde eine Testanwendung, die von TinyOS mitgeliefert wurde, auf realen Sensorknoten erfolgreich getestet, die dann mit *AvroraZ* jedoch nicht korrekt simuliert wurde. Aus diesem Grund wird für die nachfolgenden Tests Low Power Listening deaktiviert, weiterhin wird der Simulator *TOSSIM* verwendet und das im nächsten Abschnitt vorgestellte Energiemodell zur Simulation hinzugezogen.

5.2 Testmodell

Zwei Probleme für die Beweisführung ergeben sich bei der Verwendung des Simulators *TOSSIMz*. Zum einen werden durch die fehlende Simulatorunterstützung von Low Power Listening die Radio-Chips niemals abgeschaltet, zum anderen unterstützt der Simulator nur eine Sendestärke und somit nur eine feste Sendereichweite. Durch die feste Sendereichweite, müssen sich alle Knoten bei einer Single Hop Kommunikation innerhalb dieser Reichweite befinden. Hierdurch sind die Kosten für das Senden von Nachrichten an alle Knoten gleich und es entstehen keine Vorteile, Nachrichten bevorzugt an bestimmte Knoten zu verschicken. Dies wird nochmals durch das fehlende Low Power Listening verstärkt, da die Knoten gleich viel Energie bei eingeschaltetem Chip verbrauchen, egal ob dieser arbeitet oder nicht. Aus diesem Grund wird das nun folgende Modell aufgestellt.

Sowohl *TOSSIMz* als auch *AvroraZ* berechnen den Energieverbrauch, indem der Verbrauch der verschiedenen Zustände der Chips über die Zeit addiert wird. Dafür werden in *TOSSIMz* Debugnachrichten ausgegeben, die den Zeitpunkt der Zustandswechsel angeben. Diese Ausgaben werden ausgewertet und mit ihnen der Energieverbrauch berechnet. Das hier verwendete Modell greift diesen Aspekt auf. Dabei werden jedoch nicht die Zustände der Radio-Chips beachtet, sondern die Anzahl der Nachrichten gezählt, die bei der Simulation mit *TOSSIM* versendet und empfangen werden. Dazu wird die in Abbildung 5.1 dargestellte Topologie der Sensorknoten sowohl in *TOSSIM*, als auch in dem Modell verwendet.

In den nachfolgenden Testszenarien repliziert nur Knoten 1 Daten, um die Verteilung der Energiekosten und die Verteilung der Replikate besser beobachten zu können. Somit werden nur Nachrichten von Knoten 1 zu allen anderen Knoten und umgekehrt versendet. Je weiter weg die Knoten sich von Knoten 1 befinden, desto mehr Energie muss zum Versenden aufgebracht werden. Dafür besitzen die Knoten fünf Sendestärken, für die die Energiekosten schrittweise steigen. Die Kosten können neben der Verstärkung der Sendeleistung zum Beispiel auch durch eine Multi Hop-Kommunikation entstehen. Nachfolgend werden die Energiekosten in einer fiktiven Einheit E dargestellt. Die Kosten im Modell für eine Nachricht von Knoten 1 zu den anderen Knoten und umgekehrt sind in Tabelle 5.1 angegeben.

Sendestärke	Energiekosten (E)	enthaltene Knoten
S_0	0.0	0
S_1	1.0	2,6
S_2	1.4	3, 7, 8, 11, 12
S_3	1.9	4, 9, 13, 16, 17
S_4	2.5	5, 10, 14, 18, 19, 21, 22
S_5	3.2	15, 20, 23, 24
S_6	4.0	25

Tabelle 5.1: Energiekosten von Nachrichten zwischen Knoten 1 und den restlichen Knoten

Beim Low Power Listening wird der Radio-Chip regelmäßig angeschaltet und auf ankommende Nachrichten überprüft. Wenn jeder Knoten das gleiche Intervall für das Low Power Listening

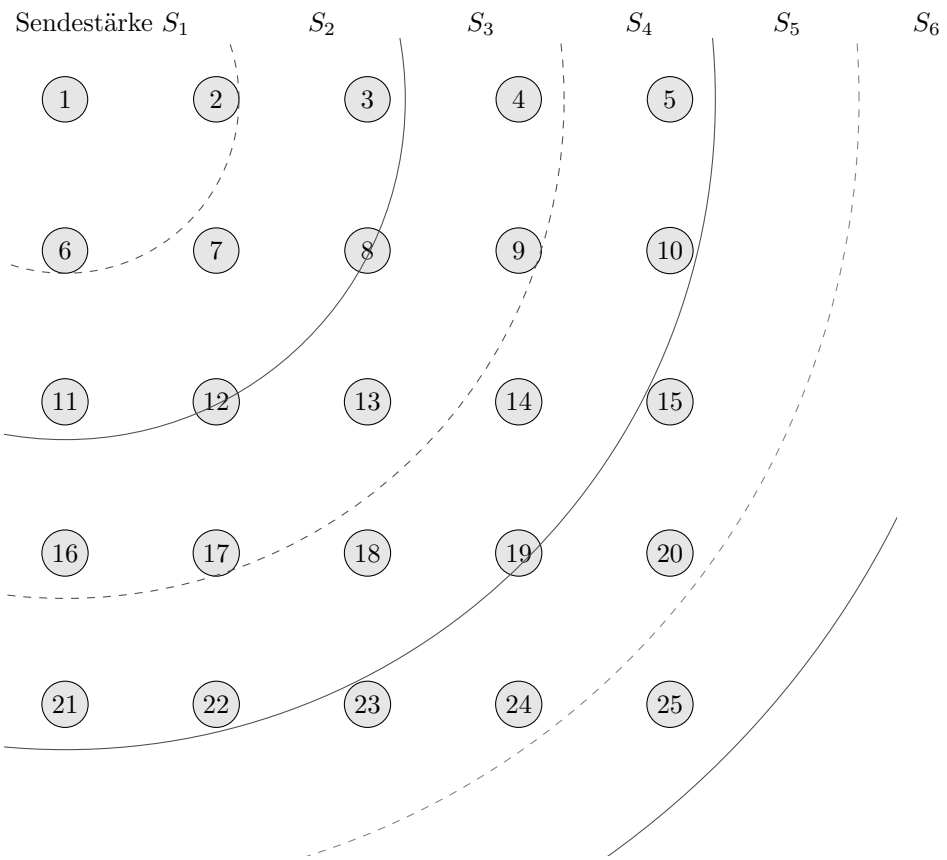


Abbildung 5.1: Topologie des Testmodells mit Sendestärken von Knoten 1

nutzt, verbraucht jeder Knoten ohne den Empfang von Nachrichten gleich viel Energie. Um annähernd zu simulieren, dass der Knoten mehr Energie verbraucht, wenn eine Nachricht für ihn bestimmt ist, wird in dem Modell zusätzlich ein Energieverbrauch von $0.5E$ für jede empfangene Nachricht berechnet. Der erhöhte Verbrauch besteht darin, dass der Chip länger angeschaltet bleiben muss, um eine für ihn bestimmte Nachricht zu empfangen.

Beispiel 9 Knoten 1 sendet zu Knoten 4 500 Nachrichten. Knoten 4 empfängt davon 490 Nachrichten und sendet an Knoten 1 150 Nachrichten. Knoten 1 empfängt alle Nachrichten von Knoten 4. Somit ist der Energieverbrauch von Knoten 1 $Energie_1 = 500 * 1.9E + 150 * 0.5E = 1025E$ und der Energieverbrauch von Knoten 4 $Energie_4 = 150 * 1.9E + 490 * 0.5E = 530E$. Das ergibt einen Gesamtverbrauch von $Energie_1 + Energie_4 = 1555$ Energieeinheiten.

5.3 Testszenarien

Die folgenden vier Tests werden mit dem Rahmenwerk durchgeführt, um den Energieverbrauch verschiedener Strategien zu überprüfen. Das zu replizierende Datenobjekt ist eine Warteschlange

mit den Funktionen *Enqueue* zum Einreihen eines Datums in die Warteschlange und *Dequeue* zum Auslesen eines Objekts aus der Warteschlange. Bei den nachfolgenden Tests wird nur die *Enqueue* Operation ausgeführt. Die *Dequeue* Operation dient in diesem Fall lediglich für die Rückverweise der Tests. In den Tests repliziert nur Knoten 1 seine Daten im Netzwerk. Die Sensordaten werden im Intervall von einer Minute ausgelesen und verteilt.

Tests 1 und Test 2 benutzen nur eine Gruppe in der Gruppenstruktur, wodurch sie eine klassische Replikationsstrategie emulieren. Test 3 und Test 4 verwenden die neuen Gruppenstrukturen. Test 2 und Test 4 benutzen außerdem die Rückverweise.

In allen Tests werden für den realen Sensorknoten 1 den physikalischen Knoten die realen Knoten folgendermaßen zugewiesen: K_0 entspricht dem realen Sensorknoten 1, K_1 wird der reale Sensorknoten 2 zugewiesen. Dies wird fortgeführt bis K_{24} gleich dem Sensorknoten 25 ist. Dadurch ist K_1 ein nahe liegender Knoten und K_{24} der entfernteste Knoten von Sensorknoten 1. Sollen die gleichen Strategien der Tests auf einem anderen Knoten angewendet werden, muss die Zuweisung entsprechend angepasst werden, so dass K_0 wieder der Knoten selber ist und K_{24} der am weitesten entfernte Knoten von dem Knoten, der die Strategien einsetzt.

Test 1

Abbildung 5.2 zeigt die Replikationsstrategie des ersten Tests. In der Gruppenstruktur der Strategie befindet sich nur die Gruppe G_0, die auf sich selber verweist und somit bei jedem Durchgang ausgewählt wird. Die Unterstruktur der Gruppe ist eine Voting-Struktur mit dem Majority Consensus Voting-Verfahren. Alle Knoten der Struktur sind gleich gewichtet, wodurch jeder Knoten bei der Erstellung der Quoren mit gleicher Wahrscheinlichkeit ausgewählt wird. Die Funktion *Enqueue* zum Verteilen der Daten benötigt ein initiales Quorum der Größe Null und ein finales Quorum der Größe 13.

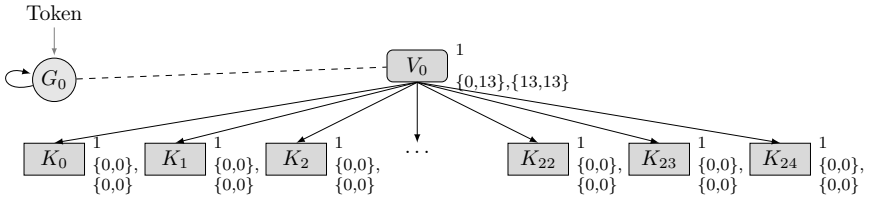


Abbildung 5.2: Replikationsstrategie Test 1

Test 2

In Abbildung 5.3 ist die Replikationsstrategie des zweiten Tests dargestellt. Die Strategie unterscheidet sich vom ersten Test nur im Rückverweis des initialen Quorums der *Enqueue* Operation auf das initiale Quorum der *Dequeue* Operation. Durch den Rückverweis werden die partiellen Historien mit den letzten Änderungen des Replikats eingesammelt und aus ihnen ein View erstellt, bevor das neue Datum an diesen gehängt und verteilt wird. Somit enthält jedes aktuelle Replikat die vollständige Historie des Objekts. Die Rückverweise können hier auch durch die direkte Angabe einer Mindeststimmenanzahl erfolgen, da immer nur die selbe Voting-Struktur verwendet wird.

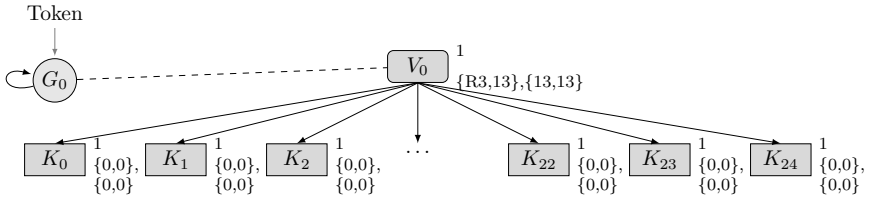


Abbildung 5.3: Replikationsstrategie Test 2

Test 3

Die Replikationsstrategie des dritten Tests ist in Abbildung 5.4 zu sehen. Sie verwendet die neu eingeführten Gruppenstrukturen. Jede Gruppe enthält als Unterstruktur direkt eine Voting-Struktur, die jeweils auf dem Majority Consensus Voting-Verfahren basiert. Alle Kanten innerhalb der Voting-Strukturen sind gleich gewichtet, wodurch jeder Knoten mit gleicher Wahrscheinlichkeit zur Bildung eines Quorums ausgewählt wird.

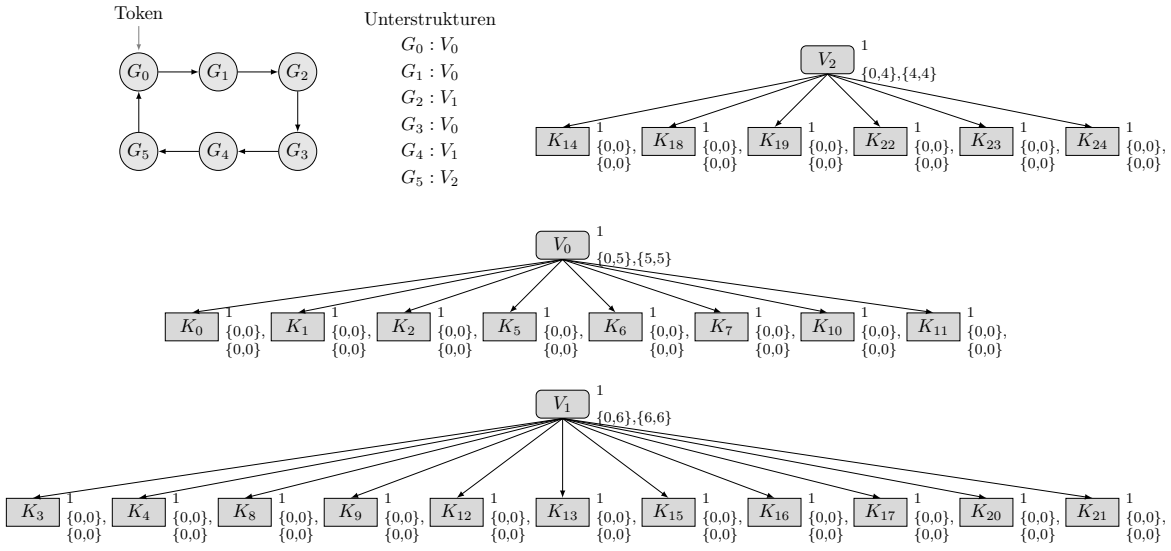


Abbildung 5.4: Replikationsstrategie Test 3

Durch die Gruppenstrukturen wird zweimal die Voting-Struktur mit dem Wurzelknoten V_0 und danach die Voting-Struktur mit dem Wurzelknoten V_1 ausgewählt. Darauf wird dann erneut die Struktur mit V_0 und dann noch einmal die Struktur mit V_1 benutzt. Zuletzt wird die Voting-Struktur mit dem Wurzelknoten V_2 zur Bildung der Quoren genutzt, bis dann die Reihenfolge von vorne durchlaufen wird. Damit wird in einem Durchgang die Voting-Struktur mit V_0 dreimal, die Struktur mit V_1 zweimal und die Struktur V_2 einmal verwendet.

Die Struktur mit V_0 enthält dabei, wie in Abbildung 5.5 auf der nächsten Seite verdeutlicht, die realen Sensorknoten, die nahe an Knoten 1 ausgelegt sind. Die Struktur mit V_1 enthält die

Knoten mit einer mittleren Entfernung zu Knoten 1 und die Struktur mit V_2 beinhaltet die am weitesten entfernten Knoten.

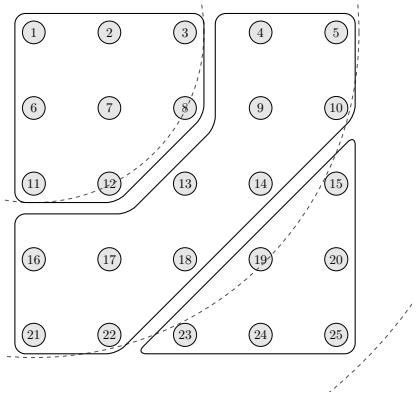


Abbildung 5.5: Einteilung der Knoten in Energiebereiche für Test 3/4

Test 4

In Abbildung 5.6 wird die Strategie des vierten Tests dargestellt. Sie unterscheidet sich zum dritten Test nur in der Verwendung der Rückverweise, wodurch ein initiales Quorum der *Enqueue* Operation aus einem initialen Quorum der *Dequeue* Operation der vorher verwendeten Voting-Struktur besteht. Ohne die Rückverweise ist in jeder Voting-Struktur nur ein Teil des Datenobjekts repliziert. Durch Verwendung der Rückverweise befindet sich jedoch in der zuletzt benutzten Voting-Struktur das vollständige Objekt.

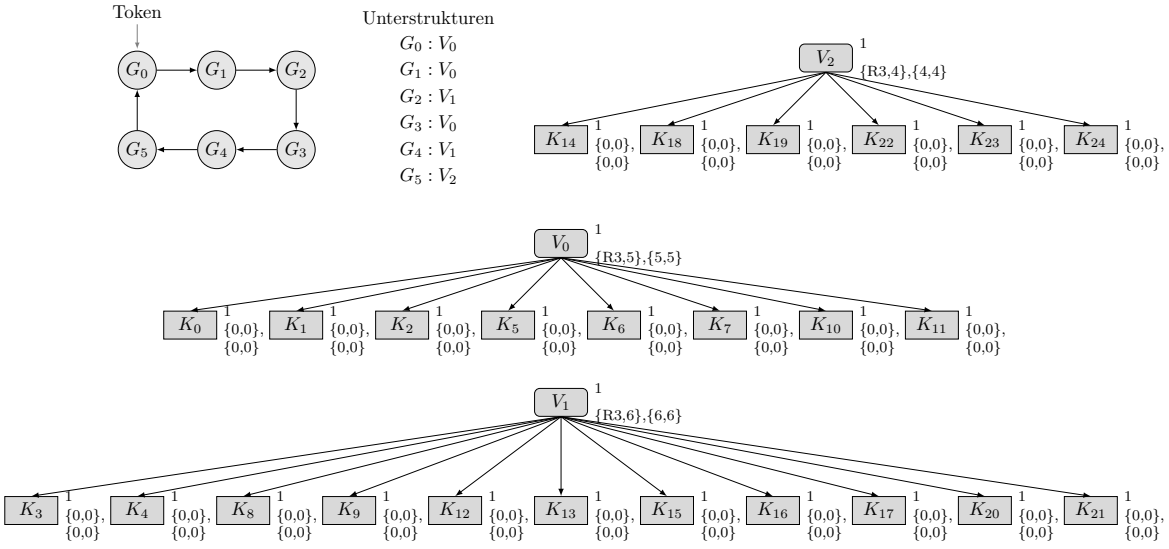


Abbildung 5.6: Replikationsstrategie Test 4

5.4 Annahmen zum Energieverbrauch

Da Test 1 keine Rückverweise besitzt und das initiale Quorum der *Enqueue* Operation leer ist, wird dieser Test vermutlich weniger Energie als Test 2 verbrauchen. Dasselbe trifft auf Test 3 und Test 4 zu, so dass Test 3 weniger Energie als Test 4 verbrauchen wird. Weiter ist der Energieverbrauch von Test 3 wahrscheinlich geringer als von Test 1 und der Verbrauch von Test 4 geringer als von Test 2, da diese durch die Gruppenstrukturen nahe kostengünstigere Knoten bei der Replikation bevorzugen und gleichzeitig die Quoren in den Strukturen, trotz des gleichen Verfahrens, kleiner sind.

Aus diesen Vermutungen ergibt sich, dass Test 3 auch weniger Energie als Test 2 verbrauchen wird. Trotz der Gruppenstrukturen muss angenommen werden, dass Test 4 mehr Energie verbrauchen wird, als Test 1, da durch die Rückverweise bei Test 4 für jede Replikation ein initiales Quorum größer Null gebildet werden muss, das zusätzliche Kosten verursacht. Besonders müssen durch die Rückverweise partielle Historien eingesammelt werden, die zum eigentlichen Zwei-Phasen-Commit Protokoll zusätzliche Nachrichten erfordern. Es ergibt sich aus den Überlegungen folgender Energieverbrauch: Test 3 < Test 1 < Test 4 < Test 2.

5.5 Ergebnisse

Jedes Testszenario wurde 20mal mit einer Simulationszeit von 150 Minuten mit dem Rahmenwerk simuliert. Die Tabelle 5.2 enthält die Mittelwerte der entstandenen Energiekosten und die Mittelwerte der Anzahl an Knoten, die an allen Replikationen teilgenommen haben. Bei Test 1 und 2, als auch bei Test 3 und 4 ist zu sehen, dass jeweils ungefähr gleich viele Knoten an den Replikationen beteiligt sind. Dabei sind die Energiekosten von Test 2 zu Test 1 und von Test 4 zu Test 3 erheblich höher. Das zeigt, dass die Rückverweise erhebliche Mehrkosten verursachen.

Test	∅ der Energie	∅ an Knoten der Verteilungen
1	30920.215 E	1937.000
2	490955.205 E	1933.100
3	10043.440 E	770.000
4	137463.000 E	766.150

Tabelle 5.2: Durchschnitt der Energiekosten und Anzahl beteiligter Knoten an allen Replikationen

Nachfolgend sind die Ergebnisse der einzelnen Tests genauer aufgeschlüsselt, um die Energieverteilung und die Verteilung der Replikate innerhalb der Tests zu beurteilen. Es ist dabei zu beachten, dass der Energieverbrauch der einzelnen Knoten sich nur auf eine Single Hop-Kommunikation mit verschiedenen Sendestärken bezieht, wie sie in dem Modell zur Simulation beschrieben ist. Bei einer Multi-Hop Kommunikation würde sich die Energie auf mehrere Knoten verteilen. Das bedeutet, dass Knoten die weit vom Empfänger entfernt sind und selten angesprochen werden zwar hohe Kosten verursachen, diese Kosten aber über die Strecke der Nachricht

verteilt werden. Muss die Kommunikation also immer über einen Knoten verlaufen, verbraucht dieser erheblich mehr Energie, auch wenn er sich direkt neben dem Sender befindet.

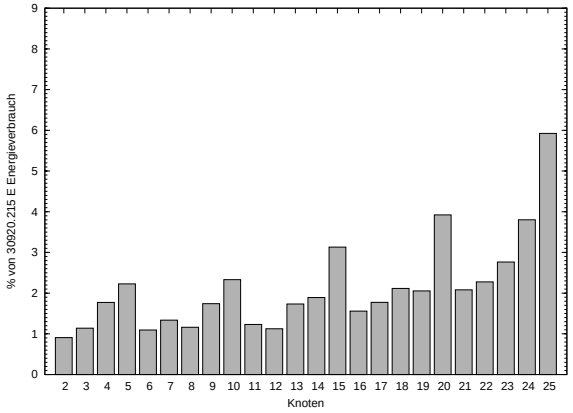
Die Verteilung des Energieverbrauchs und die Verteilung der Replikate auf den Knoten innerhalb der Tests sind der Tabelle 5.3 zu entnehmen. Die Angaben sind jeweils in Prozent der durchschnittlich verbrauchten Energie beziehungsweise der durchschnittlich beteiligten Knoten an allen Replikationen aus Tabelle 5.2 auf der vorherigen Seite angegeben. Zur einfacheren Übersicht, sind die Werte nachfolgend nochmal als Diagramme in den Abbildungen 5.7 auf der nächsten Seite und 5.8 auf Seite 72 dargestellt. Dabei wurde bei den Diagrammen zum Energieverbrauch der Knoten 1 weggelassen, da dieser in den Tests durch seine Bedeutung als Koordinator der Replikationen überdurchschnittlich viel Energie verbrauchte und die Lesbarkeit der Diagramme dadurch eingeschränkt würde.

Knoten	**% der Energie in E**				**% der Verteilungen**			
	Test 1	Test 2	Test 3	Test 4	Test 1	Test 2	Test 3	Test 4
1	48.901	36.442	51.483	48.908	3.469	3.727	5.811	5.873
2	0.908	1.080	1.798	1.631	3.696	4.172	6.129	6.128
3	1.138	1.307	2.207	2.758	3.626	3.840	5.707	6.082
4	1.772	1.870	1.713	1.570	4.323	3.543	3.649	3.341
5	2.228	2.937	2.039	2.338	4.055	4.376	3.253	3.785
6	1.094	1.062	1.865	2.009	4.470	4.247	6.376	5.684
7	1.335	1.418	1.823	2.111	4.370	4.099	4.980	5.671
8	1.162	1.472	2.316	2.309	3.722	3.915	5.987	6.467
9	1.741	1.984	1.468	1.738	4.207	3.972	3.116	3.791
10	2.332	2.801	2.599	2.161	4.279	3.944	4.123	3.498
11	1.230	1.542	2.632	2.758	3.931	4.358	6.616	6.291
12	1.125	1.335	2.498	2.657	3.603	3.758	6.441	5.866
13	1.733	1.886	1.710	1.567	4.318	4.208	3.753	3.328
14	1.892	2.488	2.248	2.298	3.680	4.089	3.779	3.765
15	3.130	4.012	1.433	1.564	4.654	3.915	1.987	2.003
16	1.558	2.202	1.254	1.762	3.861	4.089	2.688	3.576
17	1.772	2.049	1.503	1.670	4.357	4.130	3.149	3.752
18	2.114	2.768	2.080	2.111	4.078	4.151	3.506	3.197
19	2.055	2.927	1.129	1.505	3.683	3.716	1.954	2.120
20	3.921	4.147	1.904	1.941	3.843	3.996	1.928	2.166
21	2.080	3.037	2.427	2.463	3.892	3.975	3.792	3.295
22	2.276	2.824	2.606	2.525	4.140	3.853	4.149	3.628
23	2.765	4.545	1.816	1.726	3.993	4.094	2.590	2.218
24	3.801	3.933	2.344	2.255	3.717	3.941	2.350	2.277
25	5.923	7.921	3.093	3.650	4.021	3.879	2.175	2.186

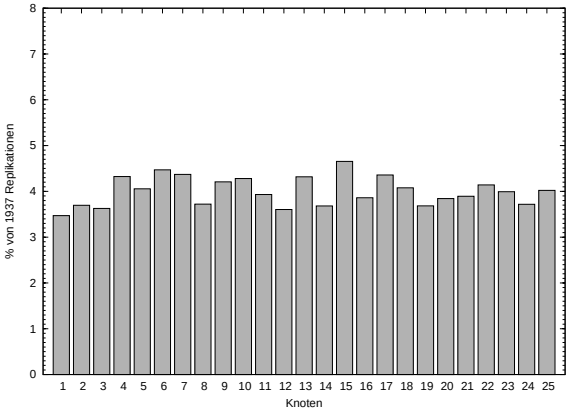
Tabelle 5.3: Energie- und Replikationsverhältnis der Knoten innerhalb der Tests

Die Diagramme 5.7a und 5.7b zeigen das Verhältnis des Energieverbrauchs der Knoten und die Verteilung der Replikate auf den Knoten des ersten Tests zueinander. Es ist zu erkennen, dass die Replikate gleichmäßig über das Netzwerk verteilt worden sind. Jeder der 25 Knoten ist an ungefähr 4% der Replikationen beteiligt. Dies macht sich jedoch beim Energieverbrauch bemerkbar. Da eine Nachricht von Knoten 25 an Knoten 1 mehr Kosten verursacht, als eine Nachricht von Knoten 6 an Knoten 1, verbraucht der Knoten 25 bei gleicher Verwendung deutlich mehr Energie.

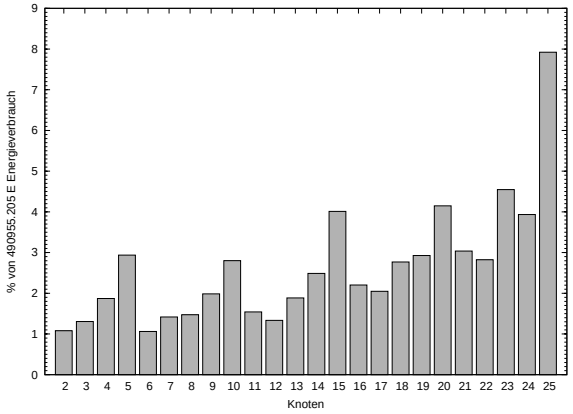
Dasselbe ist in den Diagrammen 5.7c und 5.7d zu beobachten, die die Ergebnisse von Test 2 zeigen. Hier fällt zusätzlich auf, dass die weiter entfernten Knoten noch mehr Anteile am gesamten Energieverbrauch des Tests als in Test 1 besitzen. Dies liegt an den Rückverweisen, bei denen nicht nur die Nachrichten des Zwei-Phasen-Commit-Protokolls versendet werden, sondern die Knoten außerdem noch die partielle Historie versenden. Dadurch steigen die Energiekosten, wodurch sich der Anteil der Energiekosten der zu Knoten 1 weiter entfernten Knoten erhöht. Die Replikate sind wie in Test 1 gleichmäßig verteilt.



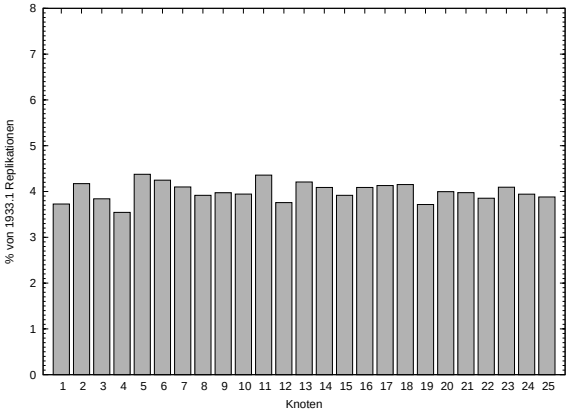
(a) Test 1: Verhältnis des Energieverbrauchs



(b) Test 1: Verhältnis der Replikationen



(c) Test 2: Verhältnis des Energieverbrauchs

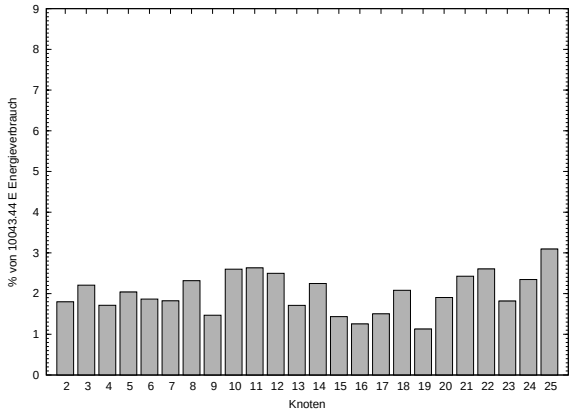


(d) Test 2: Verhältnis der Replikationen

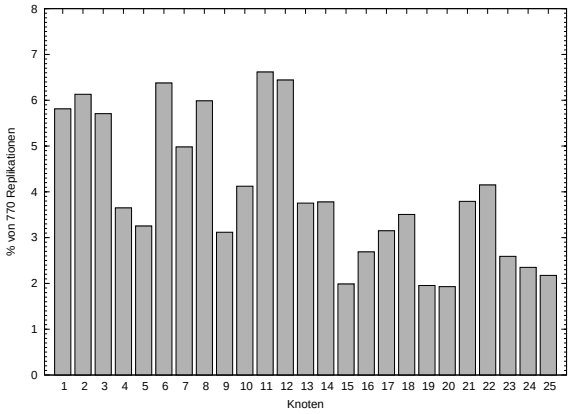
Abbildung 5.7: Ergebnisse der einzelnen Knoten von Test 1 und Test 2

Die Verteilung der Replikate von Test 3 in dem Diagramm 5.8b zeigt, dass die Replikate auf den nahen Knoten zu Knoten 1 häufiger verteilt wurden als auf den entfernten Knoten. Das ist durch die Gruppenstrukturen auch beabsichtigt. Dadurch ergibt sich ein ausgeglicheneres Verhältnis des Energieverbrauchs der Knoten, wie es in Abbildung 5.8a zu sehen ist. Es steigt zwar der Anteil am Energieverbrauch der nahe an Knoten 1 liegenden Knoten, dafür werden aber viele teure Nachrichten zu den entfernten Knoten vermieden und insgesamt Energie eingespart.

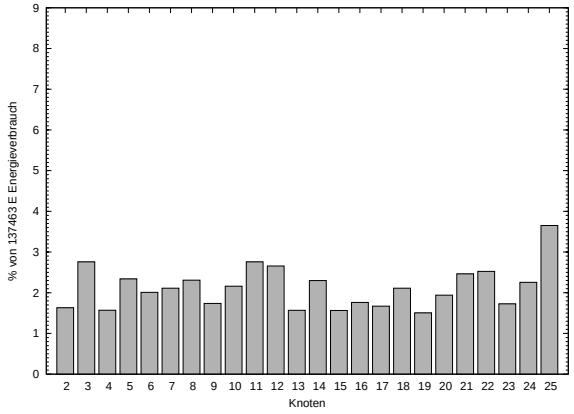
Dasselbe ist auch in den Diagrammen 5.8c und 5.8d, die die Ergebnisse des vierten Tests zeigen, zu sehen.



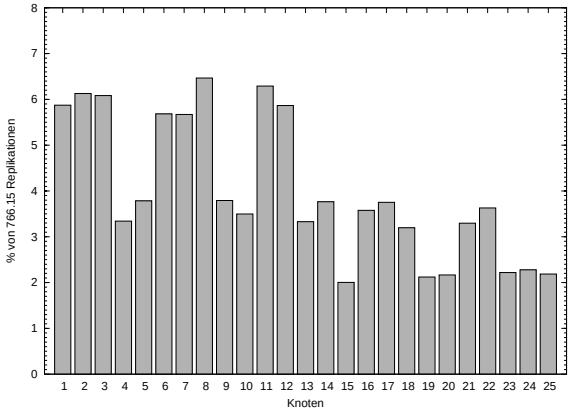
(a) Test 3: Verhältnis des Energieverbrauchs



(b) Test 3: Verhältnis der Replikationen



(c) Test 4: Verhältnis des Energieverbrauchs



(d) Test 4: Verhältnis der Replikationen

Abbildung 5.8: Ergebnisse der einzelnen Knoten von Test 3 und Test 4

Aus der Tabelle 5.2 auf Seite 69 ist abzulesen, dass die Annahmen aus Abschnitt 5.4 zutreffen. Zur Verdeutlichung wurden die durchschnittlichen Energiekosten aller Tests als Gesamtenergieverbrauch über alle Tests addiert und die einzelnen Energieverbräuche der Tests als Prozent vom Gesamtenergieverbrauch in die Tabelle 5.4 eingetragen. Zur Verdeutlichung sind die Werte in Abbildung 5.9 noch einmal grafisch dargestellt.

Test	% von Gesamtenergie (669381.860 E)
1	4.619
2	73.344
3	1.500
4	20.535

Tabelle 5.4: Testvergleich

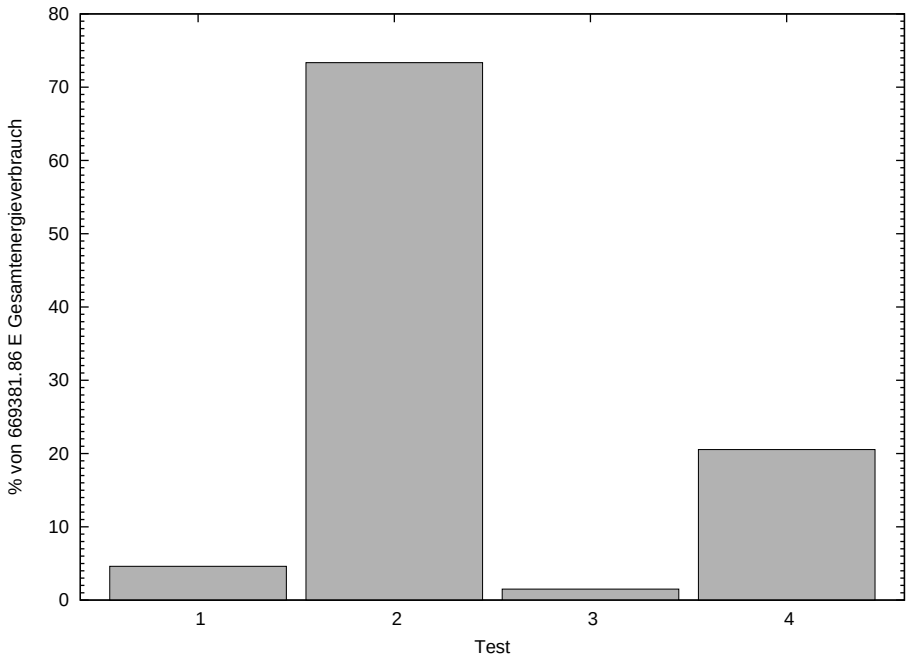


Abbildung 5.9: Gegenüberstellung der Energieverbräuche der Tests

Es ist zu sehen, dass Test 2 im gegebenen Modell dreimal so viel Energie verbraucht als Test 4. Dieser hohe Energieverbrauch entsteht zum einen durch die größeren Quoren. So müssen bei den Rückverweisen mehr Knoten angesprochen werden, die die partiellen Historien übermitteln, was den Verbrauch durch die erhöhte Anzahl der benötigten Nachrichten steigert. Zum anderen werden ohne die Gruppenstrukturen, wie vorher beschrieben, beliebige Sensorknoten ausgewählt, die mehr Energie durch teurere Nachrichten verbrauchen. Durch die Auswahl beliebiger Knoten und durch die größeren Quoren ergibt sich auch, dass Test 1 dreimal mehr Energie als Test 3 benötigt. Durch die Gruppenstrukturen werden die Quoren bei Test 3 nicht nur kleiner, sondern es werden gezielt Knoten ausgewählt, mit denen günstig Nachrichten ausgetauscht werden können.

Durch die kleinere Anzahl an Knoten in den Voting-Strukturen bei Test 3 und 4, sind diese Strategien anfälliger für Ausfälle von Knoten. So kann die Voting-Struktur von Test 1 noch mit fünf ausgefallenen Knoten korrekt arbeiten. Fallen jedoch in der Voting-Struktur mit dem Wurzelknoten V_2 von Test 3 drei Knoten aus, können keine weiteren Daten mehr repliziert

werden, so lange die Knoten ausgefallen sind und die Gruppe in der Auswahl erreicht wurde. Dieses Problem kann jedoch umgangen werden, indem die Gruppenstruktur angepasst wird und Gruppen übersprungen werden können. Hierbei ergibt sich jedoch weiterhin das Problem, dass viele wichtige Logeinträge des Objekts ohne Rückverweise verloren gehen, besonders wenn die am häufigsten genutzte Struktur nicht benutzt werden kann. Dies ist durch die erhöhte Beteiligung der Knoten an den Replikationen, wie in Test 3 zu sehen, bedingt. Dieses Problem kann eingeschränkt werden, wenn Rückverweise verwendet werden (Test 4), wodurch die vorhergegangenen Änderungen mit in die aktuelle Struktur übernommen werden. Wie jedoch in den Tests zu sehen ist, entstehen dadurch erhebliche Mehrkosten und sollten nur eingesetzt werden, wenn dies für die Operationen notwendig ist. Bei der Verwendung von Rückverweisen können, bei einem Ausfall von Knoten, Daten nicht weiter repliziert werden, falls es keinen Übersprung über die betroffene Gruppe gibt oder falls die vorhergegangene Gruppe betroffen ist und somit das initiale Quorum durch den Rückverweis nicht gebildet werden kann. Beziehen sich die Ausfälle nicht nur auf eine Voting-Struktur des neuen Konzepts, sondern auf alle, können in Test 3 zwei Knoten von V_2, drei Knoten von V_0 und fünf Knoten von V_1 ausfallen und das System könnte noch korrekt arbeiten. Trotz der Einschränkungen durch Ausfälle von Knoten, sind die in Abschnitt 5.4 auf Seite 69 getroffenen Annahmen zum Energieverbrauch bestätigt worden.

Kapitel 6

Zusammenfassung und Ausblick

Replikationsstrategien werden dazu verwendet, Daten an mehreren Orten zu speichern, um sie trotz Fehler und Ausfälle auslesen zu können. Da die Wahrscheinlichkeit eines Ausfalls einzelner Sensorknoten in einem Sensornetz hoch ist, liegt es nahe, Replikationsstrategien zu verwenden, falls die Daten nicht permanent von einer Basisstation entgegen genommen werden. Durch die Verwendung einer Replikationsstrategie ist jedoch ein erhöhter Kommunikationsaufwand und damit Energieaufwand nötig. Bei der Kommunikation ist zusätzlich die Entfernung der Knoten zueinander von Relevanz, da eine Nachricht zu einem weit entfernten Knoten mehr Energiekosten verursacht, als zu einem nahe gelegenen. Klassische Replikationsstrategien für drahtgebundene Systeme lassen diese Aspekte weitgehend außer acht. Die Strategien müssen also an die Anforderungen von Sensornetzen angepasst werden, um die Energiekosten zu berücksichtigen.

Durch das in dieser Arbeit vorgestellte Konzept können Knoten gezielt nach Energiekosten der Nachrichten gruppiert und bevorzugt verwendet werden, wodurch Energie eingespart werden kann, so dass das Sensornetz länger intakt bleibt. Dazu wurde das General Structured Voting-Verfahren um eine zusätzliche Struktur erweitert. Außerdem wurde das Verfahren an das General Quorum Consensus-Verfahren angepasst, wodurch beliebige Datentypen der Replikate und beliebige Operationen, statt der herkömmlichen Lese-/Schreiboperationen, in den Strategien verwendet werden können. Dadurch können die Replikate und ihre Funktionen besser auf die Anwendungen abgestimmt werden.

Die Ergebnisse der Simulation zeigen, dass das entwickelte Konzept Energie einsparen kann, jedoch nicht generell für alle Anwendungsgebiete geeignet ist. So muss unter anderem abgewogen werden, wie groß die Wahrscheinlichkeit von Ausfällen ist und ob die Operationen die replizierten Daten verarbeiten. Falls die Rückverweise des Konzepts verwendet werden, kann der Ausfall einer ganzen Unterstruktur durch Überspringen dieser Struktur gut verkraftet werden, da das Datenobjekt in der zuletzt verwendeten Struktur vollständig vorhanden ist. Die Rückverweise verursachen jedoch sehr hohe Extrakosten und sollten wenn möglich vermieden werden. Dies ist möglich, wenn die Operationen nicht auf dem aktuellen Stand des Datenobjekts arbeiten, sondern nur Informationen hinzufügen. Ohne die Rückverweise ist in jeder Voting-Struktur der Replikationsstrategie jedoch nur ein Teil des Datenobjekts vorhanden und ein kompletter Ausfall einer Struktur wäre fatal, denn dadurch wären Teile des Datenobjekts nicht mehr verfügbar.

Mit dem entwickelten Rahmenwerk, das das Konzept implementiert, können sowohl neue als auch klassische Strategien getestet und gegenübergestellt werden, um den Nutzen des Konzepts zu überprüfen und die Vor- und Nachteile zwischen verschiedenen Strategien abzuwägen.

Ein Ziel zukünftiger Ausarbeitungen kann sein, neue Strategien zu entwickeln und zu zeigen, wie robust diese gegen Ausfälle sind, wie viel Energie eingespart werden kann und wie hoch die Verfügbarkeit der Daten ist. Durch den prototypischen Charakter des Rahmenwerkes sind noch viele Verbesserungen möglich. Damit das Rahmenwerk auf realen Sensorknoten eingesetzt werden kann, muss beispielsweise der Speicherverbrauch verringert werden. Die Single Hop-Kommunikation könnte für den Einsatz großflächiger Sensornetze durch eine Multi Hop-Kommunikation ersetzt werden. Außerdem kann der Austausch des Zwei-Phasen-Commits durch andere Protokolle auf Energieeinsparungen getestet werden. Vor allem eine Optimierung des Versands der Historieneinträge könnte den Energieverbrauch senken. Die Übertragung des Konzepts auf das Dynamic General Structured Voting-Verfahren ermöglicht es auf Knotenausfälle zu reagieren und sollte Gegenstand fortführender Arbeiten sein.

Anhang A

Handbuch

Nachfolgend wird die Installation der Entwicklungsumgebung für das Rahmenwerk beschrieben und wie das Rahmenwerk benutzt wird.

A.1 Installation

Die Installation basiert auf *Installing from SVN/GIT*[1] der TinyOS Homepage und wird für das Betriebssystem Ubuntu 10.04 in der 32bit Version beschrieben. Die Befehle müssen in einem Terminal ausgeführt werden.

Schritt 1: Aktiviere das Stanford TinyOs Repository

```
$ sudo echo "deb http://tinyos.stanford.edu/tinyos/dists/ubuntu lucid main"
>> /etc/apt/sources.list

$ sudo apt-get update
```

Schritt 2: Installiere den nesC Compiler

```
$ sudo apt-get install nesc
```

Schritt 3: Installiere die Crosstools für AVR Chips

```
$ sudo apt-get install avr-binutils-tinyos msp430-gcc-tinyos msp430-libc-tinyos
```

Schritt 4: Lade die TinyOS-2.x Quellen aus dem Git Repository herunter

```
$ cd ~
$ mkdir -p local/src
$ cd local/src
```

1 http://docs.tinyos.net/index.php/Installing_from_SVN/GIT (25.11.2010)

```
$ git clone git://hinrg.cs.jhu.edu/git/tinyos-2.x.svn
$ cd tinyos-2.x.svn
$ git checkout -b 20090716 91725bfbd
```

Durch den letzten Befehl wird die Version vom 16.07.2009 als Branch erzeugt und geladen.

Schritt 5: Installiere Zusatzprogramme

Dieser Schritt kann übersprungen werden, wenn die Programme bereits vorhanden sind.

```
$ sudo apt-get install automake
$ sudo apt-get install g++
$ sudo apt-get install build-essential python-dev swig python-pygame
```

Schritt 6: Kompiliere die TinyOS-Werkzeuge

```
$ cd tools
$ ./Bootstrap
$ ./configure --prefix=$HOME/local
$ make all
$ make install
```

Ein paar *Enters* sind bei den letzten beiden Kommandos nötig.

Schritt 7: Variablen setzen

Die folgenden Linien müssen in die .bashrc im Homeverzeichnis kopiert werden.

```
export PATH=$HOME/local/bin:$PATH
export TOSROOT=$HOME/local/src/tinyos-2.x.svn
export TOSDIR=$TOSROOT/tos
export MAKERULES=$TOSROOT/support/make/Makerules
export CLASSPATH=$TOSROOT/support/sdk/java/tinyos.jar:.
export PYTHONPATH=.:$TOSROOT/support/sdk/python:$PYTHONPATH
export PATH=$HOME/local/src/tinyos-2.x.svn/support/sdk/c:$PATH
```

Damit die gesetzten Variablen im Terminal verfügbar sind, muss es einmal neu gestartet werden.

Schritt 8: Kopiere das Rahmenwerk

Das Rahmenwerk kann an einen beliebigen Ort von der beiliegenden CD kopiert werden.

A.2 Benutzung

Zuerst muss in den Ordner des Rahmenwerkes gewechselt werden. Um die Simulation aus Kapitel 5 zu wiederholen wird folgender Befehl mit dem unveränderten Rahmenwerk ausgeführt:

```
$ ./do.sh -auswertung
```

Die Ergebnisse der Simulation sind in *Tests/* zu finden.

Um das Rahmenwerk mit neuen Strategien zu testen, müssen die folgenden Schritte durchgeführt werden:

Schritt 1: Replikationsstrategie übertragen

In die Datei *src/VotingStructureC.nc* muss die Replikationsstrategie, wie in Abschnitt 4.1.3 beschrieben, übertragen werden.

Schritt 2: Anpassen der Einstellungen

In der Datei *src/settings.h* muss die Anzahl der Knoten und der Funktionen angepasst werden. Außerdem können hier allgemeine Einstellungen, wie die Abtastrate der Sensoren, festgelegt werden.

Schritt 3: Konfiguration der Simulation anpassen

In der Datei *simulation.py* muss die Anzahl der zu startenden Knoten angepasst werden. Hier kann außerdem die Simulationsdauer bestimmt werden. Die Topologie der Sensorknoten wird in *topologies/topo.txt* angegeben.

Schritt 4: Kompilieren und Simulieren

Nachdem alle erforderlichen Änderungen durchgeführt wurden, kann das Rahmenwerk mit folgendem Befehl für die Simulation kompiliert werden:

```
$ make SIM=1 micaz sim
```

Danach kann die Simulation mit

```
$ ./simulation.py
```

gestartet werden. Um die Daten auszuwerten, können die Befehle aus dem Shell-Skript *do.sh* verwendet werden.

Literaturverzeichnis

[BS95] Borghoff, Uwe ; Schlichter, Johann: *Rechnergestuetzte Gruppenarbeit - Eine Einfuehrung in Verteilte Anwendungen.* Berlin : Springer Verlag, 1995. – ISBN 3540581197

[Dad96] Dadam, Peter: *Verteilte Datenbanken und Client/Server-Systeme.* Berlin : Springer Verlag, 1996. – ISBN 3–540–61399–4

[Her86] Herlihy, Maurice: A Quorum-Consensus Replication Method for Abstract Data Types. In: *ACM Trans. Comput. Syst.* 4 (1986), Nr. 1, S. 32–53. http://dx.doi.org/http://doi.acm.org/10.1145/6306.6308. – DOI http://doi.acm.org/10.1145/6306.6308. – ISSN 0734–2071

[IST10] Iakab, Kinga K. ; Storm, Christian ; Theel, Oliver E.: Consistency-Driven Probabilistic Quorum System Construction for Improving Operation Availability. In: Kant, Krishna (Hrsg.) ; Pemmaraju, Sriram V. (Hrsg.) ; Sivalingam, Krishna M. (Hrsg.) ; Wu, Jie (Hrsg.): *ICDCN* Bd. 5935, Springer, 2010 (Lecture Notes in Computer Science). – ISBN 978–3–642–11321–5, 446–458

[KW05] Karl, Holger ; Willig, Andreas: *Protocols and Architectures for Wireless Sensor Networks.* John Wiley & Sons, 2005. – ISBN 0470095105

[LG09] Levis, Philip ; Gay, David: *TinyOS Programming.* New York, NY, USA : Cambridge University Press, 2009. – ISBN 0521896061, 9780521896061

[MR03] Mattern, Friedemann ; Römer, Kay: Drahtlose Sensornetze. In: *Informatik Spektrum* 26 (2003), Nr. 3, 191–194. http://link.springer.de/link/service/journals/00287/bibs/3026003/30260191.htm

[MRW97] Malkhi, Dahlia ; Reiter, Michael ; Wright, Rebecca: Probabilistic Quorum Systems. In: *PODC '97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing.* New York, NY, USA : ACM, 1997. – ISBN 0–89791–952–1, S. 267–273

[NHHT03] Niemann, Heiko ; Hasselbring, Wilhelm ; Hülsmann, Michael ; Theel, Oliver E.: Realisierung eines adaptiven Replikationsmanagers mittels J2EE-Technologie. In: Weikum, Gerhard (Hrsg.) ; Schöning, Harald (Hrsg.) ; Rahm, Erhard (Hrsg.): *BTW* Bd. 26, GI, 2003 (LNI). – ISBN 3–88579–355–5, S. 443–452

[Peu10] Peuser, Christoph: *Konzeption und Implementierung einer generalisierten Datensenke für Replikationsstrategien in Sensornetzwerken*, Carl von Ossietzky Unversität Oldenburg, Diplomarbeit, November 2010. – Die Arbeit läuft parallel zu dieser Arbeit

[Rah94] Rahm, Erhard: *Mehrrechner-Datenbanksysteme: Grundlagen der verteilten und parallelen Datenbankverarbeitung.* Bonn : Addison-Wesley, 1994 `http://dbs.uni-leipzig.de/buecher/mrdbs/mrdbs-70.html`. – ISBN 3893197028

[Sch10] Schröder, Nils: *Entwurf und Implementierung eines Sensornetzwerks zur Erfassung eines Temperaturprofils in Wohn- und Büroräumen*, Carl von Ossietzky Unversität Oldenburg, Diplomarbeit, Mai 2010

[SMZ07] Sohraby, Kazem ; Minoli, Daniel ; Znati, Taieb: *Wireless Sensor Networks: Technology, Protocols, and Applications.* John Wiley & Sons, 2007. – ISBN 978047173002

[Sto06] Storm, Christian: *Konzeption und Implementierung eines Rahmenwerkes für adaptiv-dynamische Replikationsstrategien*, Carl von Ossietzky Unversität Oldenburg, Diplomarbeit, Januar 2006

[The02] Theel, Olver E.: *Replikation in verteilten Systemen.* Vorlesung, 2002. – Carl von Ossietzky Universität

Index

V

W