AG VERNETZTE SYSTEME
FACHBEREICH INFORMATIK
AN DER TECHNISCHEN UNIVERSITÄT
KAISERSLAUTERN

PROJEKTARBEIT

BBQRT - EIN QOS-ROUTING-VERFAHREN FÜR MOBILE AD-HOC-NETZWERKE

Martin Birtel

---

18. Dezember 2008

BBQRt - Ein QoS-Routing-Verfahren für Mobile Ad-Hoc-Netzwerke

Projektarbeit

Arbeitsgruppe Vernetzte Systeme
Fachbereich Informatik
Technische Universität Kaiserslautern

Martin Birtel

Tag der Ausgabe : 15. Oktober 2008
Tag der Abgabe : 18. Dezember 2008

Betreuer : Prof. Dr. Reinhard Gotzhein
Dipl.-Inf. Philipp Becker

Ich erkläre hiermit, die vorliegende Projektarbeit selbständig verfasst zu haben. Die verwendeten Quellen und Hilfsmittel sind im Text kenntlich gemacht und im Literaturverzeichnis vollständig aufgeführt.

Kaiserslautern, den 18. Dezember 2008

(Martin Birtel)

Inhaltsverzeichnis

1 Einleitung **1**

2 Related Work **3**

3 Black Burst Quality-of-Service Routing (BBQRt) **9**

3.1 QoS-Anforderungen 9

3.2 Basistechnologien 10

3.2.1 Voraussetzungen 10

3.2.2 Black Burst Frames 11

3.2.2.1 Arbitrating Encoding 12

3.2.2.2 Restricted Arbitrating Encoding 13

3.2.2.3 Cooperative Encoding 15

3.3 Algorithmus für den Routenaufbau 16

3.3.1 Phase 1 - Route Request Selection Phase 17

3.3.2 Phase 2 - Route Discovery Phase 21

3.3.3 Phase 3 - Route Selection Phase 23

3.3.4 Phase 4 - Route Establishment Phase 26

3.3.5 Paketformate 28

4 SDL-System **31**

4.1 Architektur 31

4.1.1 Routing-Tabelle 31

4.2 SDL-Signale 33

4.3 Erwünschtes Systemverhalten (MSCs) 35

4.3.1 Initiierung eines Suchvorgangs 35

4.3.2 Ankunft beim Zielknoten 37

4.3.3 Beginn von Phase 3 37

4.3.4 Abschluss des Suchvorgangs 37

4.3.5 Weiterleitung von Datenpaketen 37

5 Zusammenfassung und Ausblick **43**

A SDL-System auf CD **45**

Kapitel 1

Einleitung

Durch die Mobilität und die fehlende feste Infrastruktur sind Verbindungen in mobilen Ad-Hoc-Netzwerken (MANETs) nur innerhalb gewisser Grenzen stabil. Um trotzdem eine sinnvolle Funktionalität bieten zu können, müssen sich die Knoten in solchen Netzwerken schnell an veränderte Bedingungen anpassen können. Dies gilt insbesondere für das Routing in MANETs. Fällt ein Netzwerkknoten aus oder ist er auf Grund seiner Mobilität nicht mehr in Reichweite, werden Routing-Informationen unter Umständen ungültig und neue Kommunikationspfade müssen aufgebaut werden. Ein weiteres Problem in Funknetzen sind die starken Schwankungen der Verbindungsqualität, die durch die Entfernung der Knoten, ihre Geschwindigkeit, die Umgebungsbedingungen (z.B. Reflektionen durch Gebäude, Abschirmung, Interferenzen) und vor allem durch gegenseitige Störungen verursacht werden. Im Gegensatz zu Kabelnetzen ist es in Funknetzen für einen Sender unmöglich, Kollisionen auf dem Medium direkt zu erkennen, weshalb auf Protokollebene Vorkehrungen zur Vermeidung getroffen werden sollten. Erhöhte Mobilität macht die Energieversorgung von Knoten schwierig und stellt hohe Anforderungen an die Energieeffizienz der Netzwerkteilnehmer. Der Einsatz von Protokollen mit geringer Nachrichtenkomplexität und speziell vorgesehen Ruhezeiten kann hier maßgeblich zur Energieoptimierung beitragen.

Quality-of-Service-Mechanismen sollen Applikationen auch unter schwierigen Bedingungen, wie bspw. in MANETs, Verbindungen mit bestimmten, garantierten Eigenschaften anbieten können (z.B. einer vom Dienstnutzer geforderten maximalen Ende-zu-Ende-Verzögerung). Zu diesem Zweck müssen Informationen bezüglich der Ressourcennutzung und -verfügbarkeit in das Routing miteinbezogen werden. Bereits existierende QoS-Routing-Verfahren konzentrieren sich oftmals auf Garantien bezüglich der Bandbreite. Da es jedoch eine Fülle möglicher Metriken für die Dienstgüte in mobilen Ad-Hoc-Netzwerken gibt, führen wir ein konfigurierbares und erweiterbares System ein, das beliebige Metriken unterstützt (bespielsweise Fehlerhäufigkeit, Verzögerung, Energieanforderungen). Für jede dieser Metriken werden verschiedene QoS-Klassen definiert, die später in den Routing-Anfragen genutzt werden können.

Trotz der inhärenten Unzuverlässigkeit von mobilen Ad-Hoc-Netzwerken ist es für Anwendungen notwendig, gewisse Mindestgarantien für die Verbindungsgüte zu erhalten. Ziel unseres Algorithmus ist es, besonders in *dichten* Netzen, effizient Routen

mit harten QoS-Garantien zwischen jeweils einem Quell- und einem Zielknoten aufzubauen.

In der vorliegenden Projektarbeit stellen wir einen neuartigen QoS-Routing-Algorithmus, genannt *BBQRt* (Black Burst Quality-of-Service Routing), für mobile Ad-Hoc-Netzwerke vor. Um Dienstgüteanforderungen zu erfüllen, werden auf allen Knoten einer Route Ressourcen reserviert und erst nach Beendigung der Verbindung wieder freigegeben. Die Dauer eines Routing-Vorganges wird bei BBQRt vom maximalen Netzwerkdurchmesser und der konfigurierbaren maximalen Routenlänge bestimmt. Sie ist somit nicht abhängig von der tatsächlichen Anzahl der Knoten und garantiert selbst in Netzen mit einer hohen Knotendichte ein effizientes Routing. Um Kollisionen während der Pfadsuche zu vermeiden und deren Dauer zu reduzieren, verwenden wir spezielle kollisionsgeschützte Frames, sogenannte *Black Burst Frames*. Sie tragen zur Sequenzialisierung von Routingvorgängen und der Auflösung von Konkurrenzsituationen bei. Im Gegensatz zum herkömmlichen Flooding oder Broadcasting in dichten Funknetzwerken werden Kollisionen aufgelöst und die Nachrichtenkomplexität wird verringert.

Die vorliegende Arbeit beschäftigt sich in Kapitel 2 mit thematisch verwandten Veröffentlichungen aus dem QoS-Routing-Bereich. In Kapitel 3 wird BBQRt für das Quality-of-Service Routing in MANETs beschrieben. Kapitel 4 beschäftigt sich mit der Spezifikation von BBQRt in Form eines SDL-Systems. In Kapitel 5 erfolgt eine Zusammenfassung mit dem Ausblick auf zukünftige Arbeiten.

Kapitel 2

Related Work

In diesem Kapitel werden existierende QoS-Routing-Protokolle vorgestellt und hinsichtlich ihrer Eignung für MANETs untersucht.

Liao et al. gehen in [hLcT02] von einem TDMA-Medium aus, nutzen aber keinen Reservierungsmechanismus zur Vermeidung von Kollisionen. Der Mechanismus konzentriert sich allein auf die Garantie von Bandbreitenanforderungen innerhalb eines mobilen Ad-Hoc-Netzwerkes. Das *Hidden Station Problem* löst das Verfahren durch den Austausch von Informationen mit den Nachbarknoten. Jeder Knoten führt eine ST-, RT- und eine H-Matrix. In der ST-Matrix wird gespeichert, in welchem Slot welcher Nachbar sendet, und in der RT-Matrix sind die entsprechenden Informationen über Empfangsoperationen enthalten. Die H-Matrix enthält die direkten Nachbarn der direkten Nachbarn von x, also die Zwei-Hop-Nachbarschaftsverhältnisse. Auf einem Knoten x haben die ST- und RT-Matrix jeweils so viele Zeilen, wie x Nachbarknoten hat. Die Spaltenzahl entspricht der Anzahl von Zeitslots. Die Erläuterungen zum Algorithmus legen nahe, dass die Tabellengrößen während eines Suchvorgangs statisch sind. Wie bei unserem Verfahren wird also davon ausgegangen, dass sich die Topologie während der Suche nach einer Route nicht ändert. Die QoS-Anforderungen werden in der Zahl der benötigten Zeitslots pro Sekunde angegeben. Ein Knoten, der eine Route sucht, schickt ein *QREQ*-Paket, das unter anderem die Parameter *b, PATH* und *NH* enthält. *b* steht hier für die Anzahl der Zeitslots, also die Bandbreite, die benötigt wird. *PATH* ist der bisher gefundene, partielle Pfad einschließlich der jeweils benutzten Zeitslots. *NH* enthält eine Liste von potentiellen Nachfolgern auf dem Pfad zum Ziel. Zu jedem dieser Nachfolger enthält *NH* eine Liste von genau *b* Zeitslots, die der Absender zur Kommunikation mit diesem nutzen könnte. Die Auswahl der konkreten Slots zu jedem dieser *NH*-Einträge geschieht auf Basis der RT-, ST- und H-Matrizen. Um Schleifen zu verhindern, erhält jede Routen-Anfrage vom Initiator eine eindeutige Id, die in den zugehörigen *QREQ*-Paketen enthalten ist. Ein *QREQ*-Paket, das den Zielknoten erreicht, enthält einen kompletten Pfad inklusive der vorberechneten Slots, die benutzt werden sollen. Der Zielknoten beantwortet die Anfrage mit einem *QREP*-Paket, das rückwärts auf der erstellten Route in Richtung Quellknoten wandert. Auf diesem Weg werden bei jedem vom *QREP* besuchten Knoten die Ressourcen reserviert und die RT- und ST-Tabellen aktualisiert. Ein großer Nachteil bei der geschilderten Vorgehensweise ist, dass Kollisionen von konkurrierenden Suchanfragen nicht vermieden werden. Dies kann vor allem in

dichten Netzen zu Problemen führen. Weiterhin ist es möglich, dass eine bereits gefundene Route, die dem Quellknoten noch nicht mittels *QREP* gemeldet worden ist, durch eine konkurrierende Reservierung zerstört wird. Da die Ressourcen nicht vorreserviert werden, kann es sein, dass sie "gestohlen" werden. Es werden keine Kriterien für eine Terminierung des Verfahrens genannt. Theoretisch können sich Anfragen eines bereits abgeschlossenen Requests noch über längere, unbestimmte Zeit im Netzwerk weiterverbreiten.

Jawhar und Wu stellen in ihrer Veröffentlichung [JW04] ein Protokoll vor, das in großen Teilen auf der Idee von [hLcT02] basiert. Allerdings werden in ihrer Variante einige Verbesserungen in Hinblick auf dichte Netzwerke eingeführt. In der ursprünglichen Form des Verfahrens in [hLcT02] können im Fall von parallelen Routen-Anfragen Konkurrenzsituationen entstehen. Angenommen, ein *QREQ*-Paket hat seinen Zielknoten erreicht und wird mit einem *QREP*-Paket beantwortet, das die berechnete Route inklusive der benötigten Zeitslots beinhaltet. Eine zweite *QREQ*-Anfrage, die über einen Knoten läuft, der die erste Antwort-Nachricht noch nicht empfangen hat, verwendet in ihren Berechnungen unter Umständen Zeitslots, die eigentlich schon vergeben wurden. Erreicht nun das *QREP*-Paket den erwähnten Knoten, wird der Zeitslot reserviert. Erst beim Besuch durch das Antwort-Paket der zweiten Routen-Anfrage wird der Konflikt entdeckt. Jawhar und Wu lösen dieses Problem durch die Einführung eines zusätzlichen Slot-Zustandes. Neben *free* und *reserved* können sich Zeitslots im Zustand *allocated* befinden. Ein Zeitlot, der als *allocated* markiert ist, ist vorreserviert und wird von konkurrierenden Anfragen nicht in die Berechnung aufgenommen. Soll ein allokierter Zeitslot für die Route verwendet werden, so wird er beim Zurückwandern des *QREP*-Paketes als *reserved* markiert und damit endgültig für den Pfad reserviert.

Ein weiteres Problem, das mit dem Verfahren aus [JW04] gelöst wird, ist das *Parallel Reservation Problem*. Im Gegensatz zur obigen Situation kann dieses jedoch nicht ohne Weiteres während des Discovery-Prozesses entdeckt werden. Angenommen, es werden gleichzeitig zwei Routen erfolgreich reserviert. Zwei oder mehr Knoten auf den beiden Pfaden befinden sich in direkter Nachbarschaft, was während des Datentranfers zu Kollisionen führen kann, wenn sie den gleichen Zeitslot verwenden. Dieses Problem kann im ursprünglichen Protokoll auftreten, weil die Sende- und Empfangsaktivitäten der direkten Nachbarn während des Suchprozesses nicht verfügbar sind. Die ST- und RT-Tabellen enthalten veraltete Informationen. Um diese Kollisionen zu vermeiden, führt im Algorithmus aus [JW04] jeder Knoten periodische Benachrichtigungen aller Nachbarknoten durch. Wenn sich ein Nachbar innerhalb einer gewissen Zeitspanne nicht meldet, wird angenommen, dass er den Sendebereich verlassen hat und er wird aus den RT-, ST- und der H-Tabelle gelöscht. Zusätzlich zu diesen Updates werden die Nachbarn benachrichtigt, wenn sich lokal der Reservierungsstatus eines Zeitslots von *free* auf *allocated* bzw. von *allocated* auf *reserved* ändert. Durch diese asynchronen und vorher erwähnten synchronen Benachrichtigungen sind die Informationen über Aktivitäten der Nachbarn immer auf dem neuesten Stand. Auf diese Art und Weise wird das *Parallel Reservation Problem* weitgehend vermieden.

Neben diesen konzeptionellen Veränderungen wurden von Jawhar und Wu einige Optimierungen entwickelt, die sich durch intensive Nutzung von Soft States auszeichnen. So erhält jede Reservierung bzw. Vorreservierung in den RT- und ST-Tabellen

einen Timer, bei dessen Ablauf der zugehörige Zeitslot in den Zustand *free* zurückversetzt wird. Bei der Benutzung eines Zeitslots für den Transfer von Datenpaketen wird der entsprechende Timer auf seinen Anfangswert zurückgesetzt.

Ein weiterer Optimierungsansatz zielt auf die Minimierung der Nachrichtenkomplexität ab. Angenommen ein *QREQ*-Paket landet bei einem Knoten, auf dem nicht genug Zeitslots als *free* markiert sind, dann besteht trotzdem die Möglichkeit, dass ausreichend Zeitslots im Zustand *allocated* vorhanden sind. Diese sind zwar vorreserviert, werden aber unter Umständen nicht genutzt und sind nach dem Ablauf eines Timers wieder frei. Es kann sich demnach lohnen, die Routing-Anfrage eine gewisse Zeit zurückzuhalten und die verfügbaren Zeitslots erneut zu prüfen. Sind nach dieser Zeitspanne immer noch nicht genügend Ressourcen frei, wird das *QREQ*-Paket verworfen. Dieses Verfahren reduziert die Wahrscheinlichkeit für die Wiederholung von Suchanfragen. Statt die Suche immer wieder von Vorne zu beginnen, wird pausiert und auf der bereits berechneten Teillösung aufgesetzt.

Das große Manko der zeitlich unbegrenzten Pfad-Suche wird in [JW04] beseitigt, indem jede Anfrage mit einem *Time-To-Live*-Feld versehen und von jedem bearbeitenden Knoten um die entsprechende Zeitspanne verringert wird. Erreicht die Anfrage das Ziel, wird dieses Feld in die Antwort integriert und ebenfalls bei jeder Verarbeitung dekrementiert. Falls der Zähler den Wert 0 erreicht, wird das Paket verworfen und es werden optional der Initiator und alle an der Bearbeitung beteiligten Knoten benachrichtigt. Auf diesem Weg werden bereits vorreservierte Ressourcen explizit, anstatt durch ein Timeout, freigegeben und können schneller wiederverwendet werden. Leider belegen Jawhar und Wu ihre Arbeit nicht mit entsprechenden Messergebnissen. Es liegt keine Untersuchung zur Praxistauglichkeit vor.

Lin geht in seinem Protokoll von einem gänzlich anderen Ansatz zur Vermeidung von Kollisionen aus. Dem System [Lin01] liegt eine *CDMA-over-TDMA*-Vorgehensweise zu Grunde. Die Zeit wird in Macro-Slots fester Größe mit einer Kontroll- und einer Datenphase eingeteilt. Die Datenphase wiederum ist in Slots gegliedert, die der Übertragung von Daten dienen sollen. Die Reservierung von Slots erfolgt in der Kontrollphase. Das *Hidden Station*, das *Exposed Station* und das *Parallel Reservation Problem* werden unter Verwendung von CDMA (*Code Division Multiple Access*) umgangen. Es wird angenommen, dass in der darunter liegenden Protokollschicht eine ideale Zuweisung von paarweise orthogonalen Spreading-Codes stattfindet. Jeder Netzwerkknoten erhält also einen eindeutigen Code, dessen Verwendung sicherstellt, dass keine Kollisionen mit Nachbarn auftreten können.

Der Algorithmus aus [Lin01] zielt darauf ab, Funknetzwerke an feste Kabelnetze, wie bspw. ATM, anzubinden. Um weiterhin verlässliche Garantien für die Verbindungsqualität geben zu können, muss die *QoS*-Funktionalität auf die Funkknoten ausgedehnt werden. Dabei werden sowohl paketvermittelte Übertragung als auch *Virtual Circuits* unterstützt.

Der Algorithmus ist rein Anfragen-basiert und benötigt keinen periodischen Austausch von Topologie- oder anderen Routing-Informationen. Der Mechanismus in [Lin01] ist dem in [hLcT02] ähnlich. Benötigt ein Knoten eine Route zu einem Ziel, schickt er ein *RREQ*-Paket, das unter anderem eine eindeutige Sequenznummer, ein TTL-Feld, die benötigte Bandbreite und die Adresse des Zielknotens enthält.

Die Sequenznummer dient dazu, Schleifen in der Route zu verhindern. Empfängt ein Knoten eine Anfrage, berechnet er die bisher erreichte Bandbreite und prüft, ob er genügend freie Zeitslots besitzt, um die QoS-Anforderungen zu erfüllen und dekrementiert den TTL-Zähler. Besitzt er nicht genug freie Ressourcen oder ist der TTL-Zähler abgelaufen, so verwirft er das Paket. Andernfalls trägt er die Anzahl seiner freien Slots für die Übetragung zum Nachfolger auf der Route in das Paket ein, hängt seine Adresse an die im Paket enthaltene, partielle Route an und sendet es weiter.

Jedes *RREQ*-Paket, das beim Zielknoten ankommt, enthält eine eindeutige Route, die die QoS-Anforderungen erfüllt. Der Zielknoten merkt sich mehrere dieser Pfade und kann im Fall einer gescheiterten Reservierung den Aufbau einer Alternativroute starten. Beim Erreichen einer Anfrage beim Zielknoten schickt dieser ein *QREP*-Paket, das die berechnete Route und die zugehörige Liste von Zeitslots beinhaltet, auf dem reversen Pfad an den Quellknoten. Bei den besuchten Knoten werden die für die Route ermittelten Slots reserviert. Es kann jedoch dazu kommen, dass zwei gleichzeitige Berechnungen denselben Zeitslot auf einem gemeinsamen Knoten miteinbeziehen. Die *QREP*-Nachricht, die diesen Knoten zuerst besucht, kann den gewünschten Slot reservieren. Die zweite Route muss verworfen werden, da der benötigte Slot bereits vergeben ist. In diesem Fall werden alle Knoten auf dem Pfad, die bereits Ressourcenn reserviert haben, benachrichtigt und veranlasst diese freizugeben. Sobald der Zielknoten die sogenannte *NACK (RESERVE FAIL)*-Nachricht erhält, wählt er aus der Menge der berechneten Routen eine aus und startet einen weiteren Reservierungsvorgang. Hat das Ziel keine alternative Route mehr zur Verfügung, wird ein *NACK (NO ROUTE)* zum Quellknoten geschickt.

Wird eine Route auf Grund von Knotenausfällen oder einer Änderung der Topologie zerstört, so melden die Knoten an der Bruchstelle das Versagen weiter. Der eine schickt ein *NACK (ROUTE BROKEN)* zum Ziel-, der andere zum Quellknoten des Pfades. Knoten, die zu dieser Route gehören und diese Nachricht empfangen, geben die dazugehörigen Ressourcenn frei, verwerfen noch ausstehende Pakete und senden das *NACK (ROUTE BROKEN)* weiter. Sobald die Nachricht über das Versagen den Quellknoten erreicht, startet dieser eine neue Routing-Anfrage. Unter normalen Umständen sollen reservierte Zeitslots nach Ende einer Verbindung explizit freigegeben werden. Geschieht dies nicht, werden die Slots nach einer gewissen Zeitspanne ohne Datenverkehr automatisch freigegeben.

Die erwähnten Verfahren besitzen besonders in Hinblick auf die Verwendung in dichten Netzwerken Schwächen, die sich negativ auf die Performanz auswirken können. In den Protokollen von Liao et al. und Jawhar und Wu ist es möglich, dass Nachbarknoten gleichzeitig Leitweganforderungen stellen, die kollidieren und so zerstört werden. Beim BBQRt wird dieses Problem durch die Sequenzialisierung von Suchanfragen gelöst. Außerdem wird durch sie vermieden, dass die selben Ressourcen während eines Suchvorgangs *mehrfach* reserviert werden können, wie es unter Umständen bei Liao et al. der Fall ist. Der Versuch, die selben Zeitslots erneut zu reservieren, wird durch den Algorithmus von Lin entdeckt, allen Knoten des bisher berechneten Teilpfades signalisiert und stößt in der Regel den Aufbau einer alternativen Route an. Dies führt zu einer höheren Nachrichtenkomplexität in Konkurrenzsituationen.

Der Pfadaufbau im BBQRt besitzt vier Phasen von berechenbarer, fester Dauer und mit festgelegten Grenzen, die vom Netzwerkdurchmesser und dem vorgegebenen Parameter für die maximale Routenlänge abhängen. Die Netzwerkteilnehmer wissen deshalb genau, in welchem Zeiträumen innerhalb der Phasen Nachrichten zu erwarten sind und wann ein Abhören des Funkmediums unnötig ist. Auf diesen Informationen aufbauend kann ein effektives Energiemanagement erfolgen, das besonders in Szenarien mit erhöhter Mobilität große Vorteile bietet. Im Vergleich dazu besitzen die oben erwähnten Algorithmen entweder keine zeitliche Begrenzung der Pfadsuche (wie in [hLcT02]) oder sie erlauben die parallele Abarbeitung mehrerer Leitweganforderungen, wodurch die Vorhersage von potentiellen Pausenzeiten schwierig wird.

In den beschriebenen Veröffentlichungen wird hauptsächlich die Bandbreitenmetrik in den QoS-Anforderungen verwendet. BBQRt hingegen unterstützt durch einen allgemeineren Ansatz verschiedene QoS-Metriken und bietet durch einen Konfigurationsmechanismus die Option auf Erweiterung.

Kapitel 3

Black Burst Quality-of-Service Routing (BBQRt)

In diesem Kapitel werden unser QoS-Routing-Algorithmus für MANETs sowie die notwendigen Voraussetzungen für seine Anwendung beschrieben.

3.1 QoS-Anforderungen

Eine Metrik weist jeder Route einen Wert zu, der zur Beurteilung ihrer Qualität in Bezug auf einen bestimmten Aspekt und somit auch dem Vergleich mit anderen Routen dienen soll. Beim QoS-Routing wird zwischen verschiedenen Arten von Metriken unterschieden, die auf der Grundlage der Einzelverbindungsmetriken des Pfades berechnet werden. Bei additiven Metriken, wie beispielsweise der Gesamtverzögerung entlang eines Pfades, müssen die Metrikwerte von Verbindungen zwischen Pfadknoten addiert werden. Bei multiplikativen Metriken werden die Einzelwerte multipliziert. Eine weitere Art sind die konkaven Metriken, wie zum Beispiel die Bandbreite. Hier wird über der Gesamtheit der Einzelverbindungsmetriken minimiert.

Die Unterstützung unterschiedlicher Metriken macht in vielen Fällen die Beschaffung umfassender Informationen über den Netzwerkstatus notwendig. Probleme entstehen hierbei, wenn die Suche nach möglichen Routen *parallel* mit der Berechnung der korrespondierenden Metriken durchgeführt wird. Oftmals ist die sukzessive Ermittlung der Routenmetrik in einer Richtung (z.B. vom Quell- zum Zielknoten hin) nicht möglich und die endgültige Bewertung kann erst erfolgen, wenn alle Teilverbindungen des Pfades bekannt sind. Ein Beispiel hierfür ist die Garantie von Bandbreite in einem Funknetz mit *TDMA*. Es kann ausgehend vom Quellknoten ermittelt werden, ob auf allen Knoten des bisher berechneten Teilpfades theoretisch genug Bandbreite verfügbar ist, die Erfüllbarkeit der gestellten Anforderungen kann bis zum Erreichen des Ziels jedoch nicht entschieden werden, da diese von der Reservierung der Zeitslots abhängt. In der vorliegenden Arbeit wird aus Zeitgründen vorausgesetzt, dass lokal entscheidbare Metriken zum Einsatz kommen und die Pfadsuche auf Basis lokal vorhandener und mit den Routing-Paketen mitgeschickter Informationen durchgeführt werden kann.

In unserem System werden die QoS-Anforderungen einer Routen-Anfrage durch zwei Parameter beschrieben. Der QoS-Typ gibt die zu verwendende Metrik an. Unterstützte Metriken besitzen eine eindeutige Id und sind in die Metriktabelle jedes Knotens eingetragen. Der zweite Parameter, die QoS-Klasse, beschreibt den zahlenmäßigen Wert, der garantiert werden soll. Da die Metriken in der Regel kontinuierlich und nicht beschränkt sind, muss eine Quantisierung vorgenommen werden, die die Tabellengröße begrenzt. Um einen konkreten QoS-Wert anzufordern, wird im QoS-Klassen-Feld die Zeilennummer des Wertes in der Tabelle angegeben. Die Größe der erwähnten Tabellen muss vor Inbetriebnahme des Netzes festgelegt und Kopien der Tabellen auf jedem Knoten installiert werden. Im Folgenden werden beispielhaft QoS-Tabellen mit einer Größe von vier Einträgen für den QoS-Typ und acht für die QoS-Klasse (siehe Tabelle 3.1) aufgeführt.

Im konkreten Fall müssen die Tabellen an die Hardware und Anforderungsprofile angepasst werden, um die vorhandenen Ressourcen besser nutzen zu können. Beispielsweise würde eine zu grobe Einteilung des QoS-Wertebereichs im Fall von vielen Verbindungen mit geringen Bandbreitenanforderungen zu ungenutzten Kapazitäten führen. Andererseits kann eine Wahl von zu niedrigen Werten in der QoS-Klassentabelle bewirken, dass die Anforderungen an Verbindungen mit sehr hohem Datenaufkommen nicht beschrieben werden können. Der hier vorgeschlagene Mechanismus ermöglicht es, die QoS-Kodierung auf unterschiedliche Anwendungsszenarien auszurichten und hinsichtlich des Speicherbedarfs und der Granularität zu optimieren.

QoS-Typnummer	Metrik
0	Bandbreite
1	Verzögerung
2	Fehlerrate
3	Energie

QoS-Klassennummer	QoS-Wert
0	32 Bit/s
1	64 Bit/s
2	96 Bit/s
3	1 kBit/s
4	2 kBit/s
5	4 kBit/s
6	8 kBit/s
7	16 kBit/s

Tabelle 3.1: Beispiele für QoS-Typen- und -Klassentabelle

3.2 Basistechnologien

3.2.1 Voraussetzungen

Im folgenden setzen wir ein Funkmedium mit einer speziellen MAC-Schicht voraus: MacZ (siehe [BGK07]). MacZ arbeitet im TDMA-Modus, bei dem die Zugriffe auf das Netzwerk über Zeitslots geregelt werden. Die Zeit wird in Macro-Slots mit jeweils einer Resynchronisations- und einer Daten-Phase eingeteilt. In der Resynchronisations-Phase werden die Knoten im Netz, bis auf eine deterministische Abweichung, ticksynchronisiert. Dies geschieht durch einen in MacZ eingebauten,

verteilten Synchronisationsmechanismus auf Basis von kollisionsgeschützen Tick-Frames. Die nachfolgende Daten-Phase wird in Micro-Slots fester Länge eingeteilt, die jeweils zum Senden von Daten genutzt werden können. Durch die Ticksynchronisation kennt jeder Knoten im Netz die Startzeiten der Micro-Slots. Zur Verhinderung von Kollisionen und zur Garantie von Bandbreite können Netzwerkknoten mit Hilfe eines Reservierungsprotokolls einen oder mehrere Micro-Slots für sich beanspruchen und in der Daten-Phase nutzen. Nachbarknoten werden automatisch über den Reservierungsstatus benachrichtigt. Die Reservierung muss spätestens nach einer definierten Zeitspanne aufgefrischt werden, sonst verfällt sie. Konkurrierende Reservierungsversuche müssen von höher gelegenen Protokollschichten aufgelöst werden.

Eine weitere wichtige Voraussetzung ist die begrenzte Mobilität von Netzwerkknoten. Die Routing-Schicht geht davon aus, dass sich die Netzwerktopologie während der *Pfadsuche* nicht signifikant verändert. Nachfolgende Knotenausfälle und -bewegungen werden durch einen Timeout-Mechanismus entdeckt und eventuell auftretende Probleme können gegebenenfalls durch den Aufbau eines neuen Pfades behoben werden.

3.2.2 Black Burst Frames

Black Burst Frames sind spezielle Frames, die die kollisionsgeschützte Übertragung von Daten in einem Funknetzwerk sicherstellen. Dabei spielt der Inhalt eines einzelnen *Black Bursts* keine Rolle, lediglich sein Auftreten innerhalb eines bestimmten Zeitraums ist von Relevanz (dazu siehe auch [KdI07]). Falls alle Knoten ticksynchronisiert sind und wissen, zu welchen Zeitpunkten eine Übertragung möglich ist, können Bit-Sequenzen unter der Verwendung von *Black Bursts* folgendermaßen kodiert werden. Eine binäre 1 wird als das Vorhandensein eines *Black Bursts* in einem Zeitslot, eine 0 als dessen Abwesenheit kodiert. Senden nun allerdings mehrere Knoten gleichzeitig, kommt es bei Empfängern, die sich in deren Reichweite befinden, zu Kollisionen. Der Inhalt von *Black Burst Frames* muss allerdings nicht gelesen werden, es genügt mittels *Clear Channel Assessment* ihr Vorhandensein auf dem Medium festzustellen. Eine Kollision von *Black Bursts* führt demnach zu einer Überlappung auf dem Medium. Probleme können hier auftreten, wenn die Überlappung über die Grenzen eines Zeitslots hinausgeht. Empfänger würden im nächsten Zeitslot ebenfalls ein Signal hören und dies als 1 werten. Durch die Ticksynchronisation und die Länge der Zeitslots ist jedoch gewährleistet, dass sich *Black Burst Frames* nur bis zu einem gewissen Grad überlappen können und die zeitlichen Grenzen der Slots nicht verletzen. Effektiv bedeutet dies, dass sich gleichzeitig mittels *Black Bursts* gesendete Einsen nicht stören. Anders sieht dies im Fall von unterschiedlichen Übertragungen aus. Sendet eine Knoten eine binäre 1 und ein anderer eine binäre 0, so hört ein Empfänger in deren Reichweite eine 1. Es findet eine logische ODER-Verknüpfung der gesendeten Bits statt.

In den nächsten beiden Abschnitten folgen Erläuterungen zu den unterschiedlichen Encodings beim Versand von *Black Bursts* mit MacZ. Dabei sind insbesondere Timing-Fragen und die deterministische Dauer einer Übertragung von Interesse.

3.2.2.1 Arbitrating Encoding

Single-Hop In Netzwerken tritt häufig der Fall ein, dass mehrere Sender zur gleichen Zeit beginnen, Bit-Sequenzen zu verschicken (siehe Abbildung 3.1). Die daraus entstehenden Konkurrenzsituationen von benachbarten Funkknoten sollen derart aufgelöst werden, dass sich die dominante Bitfolge auf dem Medium durchsetzt. Dies wird im Fall von *Black Burst Frames* durch das sogenannte *Arbitrating Encoding* gewährleistet.

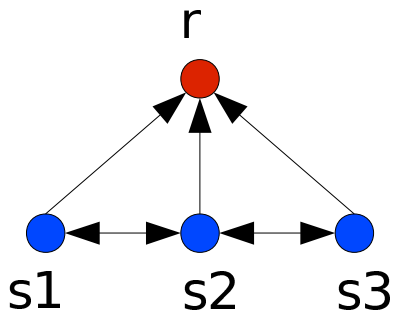

Abbildung 3.1: Konkurrierende Nachbarknoten

Beim Versenden von *Black Burst Frames* werden binäre Einsen als das Vorhandensein eines *Black Bursts* und binäre Nullen als dessen Abwesenheit innerhalb eines bestimmten Zeitraumes kodiert. Da alle Knoten ticksynchronisiert sind und die Sendevorgänge - bis auf eine geringe Verschiebung - zeitgleich stattfinden, gibt es vordefinierte Zeitpunkte, an denen Übertragungen stattfinden können. Ein Knoten, der eine Eins senden möchte, belegt das Medium mit einem *Black Burst* und ist in diesem Zeitraum nicht in der Lage die Übertragung konkurrierender Knoten festzustellen. Im Gegensatz dazu können Knoten, die eine Null „versenden“, eine vom Nachbarn geschickte, dominante Eins hören. Geschieht dies, weiß der Empfänger, dass er eine Bit-Sequenz mit niedrigerer Priorität versenden möchte und zieht sich zurück. Falls die Knoten unterschiedliche Bit-Sequenzen übermitteln, hat sich am Ende genau eine (innerhalb der 1-Hop-Nachbarschaft) durchgesetzt.

Die Kodierung der *Black Burst Frames* birgt jedoch eine Schwierigkeit: ein potentieller Empfänger ist nicht in der Lage zu entscheiden, ob gerade eine Bit-Sequenz versendet wird, die mit einer Reihe von Nullen beginnt oder ob zur Zeit nichts gesendet wird. Aus diesem Grund muss einer zu versendenden Bitsequenz beim *Arbitrating Encoding* ein *Start-of-Frame*-Bit vorangestellt werden. Dieses auf 1 gesetzte Bit markiert den Beginn der Bit-Sequenz.

Weiterleitung In der Regel soll eine mittels *Arbitrating Encoding* gesendete Bit-Sequenz alle Knoten im Netzwerk erreichen. Dies wird durch die *sofortige* Weiterleitung jedes empfangenen Bits an die Nachbarknoten erreicht. Wenn sich ein Knoten im oben beschriebenen Arbitrierungsverfahren zurückzieht, beteiligt er sich trotzdem weiter an der Datenübertragung, indem er empfangene Eins-Bits weiterleitet. Für die Gesamtdauer einer Bit-Übertragung gibt es eine deterministische Obergrenze,

die vom maximalen Netzwerkdurchmesser, aber *nicht* von der Anzahl der Knoten abhängt.

Zeitbedarf Sei d_{BB} die benötigte Zeit, um im *Arbitrating Encoding* ein Bit mittels *Black Burst Frame* an die 1-Hop-Nachbarn zu versenden. k sei die feste, den Netzwerkteilnehmern bekannte Länge der zu sendenden Bitsequenz. Dann kann die Dauer einer Übertragung mittels *Arbitrating Encoding* bei uneingeschränkter Weiterleitung und einem maximalen Netzwerkdurchmesser von $maxDiam$ wie folgt berechnet werden:

$$d_{arb} = k \cdot d_{BB} \cdot maxDiam$$

3.2.2.2 Restricted Arbitrating Encoding

Limitierte Weiterleitung In Funknetzen kann das *Hidden Station Problem* auftreten, wenn sich konkurrierende Knoten nicht direkt hören können und deshalb gleichzeitig Nachrichten verschicken. Dadurch entstehen bei einem Empfänger in deren Sendebereich Kollisionen. Das *Arbitrating Encoding* ist jedoch in bestimmten Szenarien darauf angewiesen, dass Mitbewerber die Übertragungen der Anderen bemerken. Dies soll durch eine modifizierte Variante des oben vorgestellten *Arbitrating Encoding* sichergestellt werden. Der Knoten *f*, an dem die Kollisionen auftreten können, muss die Bitfolgen, die er hört, an die Konkurrenten weiterleiten, damit das Arbitrierungsverfahren wie oben beschrieben durchgeführt werden kann. Knoten, die Daten übertragen möchten, senden zunächst ein *Start-of-Frame*-Bit (SoF-Bit), danach folgt ein *Forwarding*-Flag. Wird das SoF-Bit von einem Knoten empfangen, wartet er auf das *Forwarding*-Flag. Ist das Bit 1, dann soll die Nachricht weitergeleitet werden und der Knoten muss seinerseits ein SoF-Bit und ein *Forwarding*-Flag an die eigenen Nachbarn übertragen. Dieses ist jedoch auf 0 gesetzt, damit Knoten, die mehr als zwei Hops vom Sender entfernt sind und nicht beteiligt werden sollen (z.B. Knoten x in Abb. 3.2), die Nachricht verwerfen. Die restlichen direkten, konkurrierenden Nachbarn des Empfangsknotens f ignorieren das Flag und dürfen die empfangenen Daten verarbeiten. Abbildung 3.3 zeigt die Übertragung und Weiterleitung einer Bit-Sequenz. Der ursprüngliche Sender muss pausieren während der weiterleitende Knoten den Header - bestehend aus SoF-Bit und *Forwarding*-Flag - schickt. Erst danach folgt die bitweise Übermittlung der Daten. Der Sendeknoten versendet ein Bit, wartet eine Bitzeit, bis dieses weitergeleitet wurde und schickt dann das nächste. So wird verfahren, bis die komplette Sequenz übertragen worden ist oder der Sender die dominante Übertragung eines Konkurrenten gehört hat und sich zurückzieht.

Am Ende der Übermittlung hat sich der Knoten f die dominante Bitsequenz gemerkt und kann nun selbst als Sender agieren. Das Ende einer Sendephase muss entweder explizit signalisiert werden oder durch eine bekannte, feste Länge der gesendeten Sequenzen bestimmt sein.

Arbitrierungs-Algorithmus mit limitierter Weiterleitung

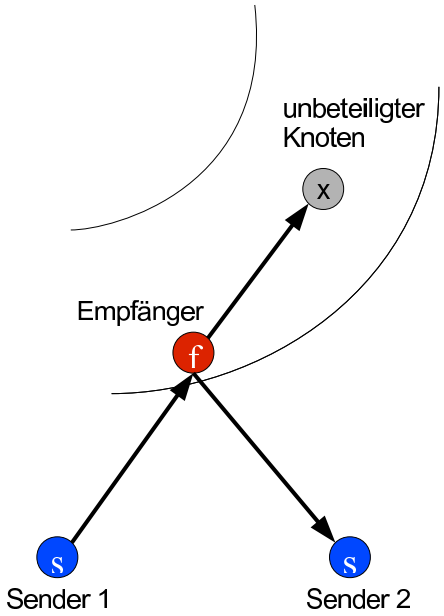

Abbildung 3.2: Konkurrierende Sendeknoten

Sender

```
// send start of frame (SoF) bit
sendBit(1)
// send forward flag
sendBit(1)

// wait two bit cycles in order to give receivers time to send
// their SoF and forward flag bits
sendBit(0)
sendBit(0)

// send own bit sequence
for (i := 0; i < seqLength; i++)
    if (mySequence[i] = 0)
        // we are sending a 0 => listen for dominant
        // bit sent by direct neighbors
        if (receiveBit() = 1)
            // dominant transmission detected
            backoff()
        endif
    else
        sendBit(mySequence[i])
    endif

    // listen for forwarded dominant transmission
    if (receiveBit() = 1)
```

```
            // dominant transmission detected
            backoff()
        endif
    endfor
```

Empfänger

```
    if (startOfFrame != undef)
        bit := nextBit()
        if (bit = 0)
            // message has been forwarded => drop it
            break
        endif

        // message comes from a potential predecessor => forward it
        // send start of frame (SoF) bit
        sendBit(1)
        // send forward flag
        sendBit(0)
        for (i := 0; i < seqLength; i++)
            bit := receiveBit()
            appendBit(message, bit)
            sendBit(bit)
        endfor
    endif
```

Zeitbedarf Sei d_{BB} die Zeit, um im *Arbitrating Encoding* ein Bit mittels eines *Black Bursts* an die 1-Hop-Nachbarn zu versenden. k sei die feste, dem Empfänger bekannte Länge der zu übertragenden Bit-Sequenz. Ein Sender benötigt zwei Bitzeiten, um das SoF- und das *Forward*-Bit zu versenden. Anschließend macht er eine Pause von zwei weiteren Bitzeiten, in der der oder die Empfänger ihrerseits ein SoF- und ein *Forward*-Bit versenden. Anschließend überträgt der Sender k-mal ein Bit, das vom Empfänger an dessen direkte 1-Hop-Nachbarn weitergeleitet wird.

Insgesamt werden also

$$d_{restrarb} = (4 + 2k)d_{BB}$$

Zeiteinheiten für den Versand einer Sequenz aus k Bits benötigt (siehe Abbildung 3.3 für eine beispielhafte Datenübertragung mit Weiterleitung).

3.2.2.3 Cooperative Encoding

Das *Cooperative Encoding* wird verwendet, um eine Nachricht an alle Knoten im Netzwerk zu schicken. Dabei wird die Bit-Sequenz, anders als beim *Arbitrating Encoding*, nicht Bit für Bit, sondern als *ein* Frame an die Nachbarknoten gesendet. Erst, wenn die Nachricht komplett beim Empfänger angekommen ist, wird diese weitergeleitet. Knoten, die Daten übertragen möchten, nehmen im Gegensatz zum

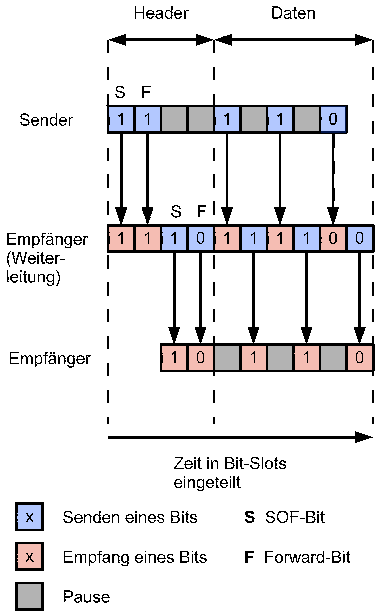

Abbildung 3.3: Limitierte Weiterleitung beim *Arbitrating Encoding*

Arbitrating Encoding keine Rücksicht auf Sequenzen mit höherer Priorität. Kollisionen müssen deshalb von vornherein ausgeschlossen werden, indem alle gleichzeitig sendenden Knoten die selben Daten verschicken.

Eine vom Quellknoten gestartete Übertragung wird mit einem Rundenzähler versehen und weiter versendet. Alle Knoten, die die Originalnachricht empfangen haben, werden in der nächsten Runde aktiv, erhöhen den Rundenzähler und verschicken die empfangene Sequenz an ihre Nachbarn. Knoten, die bereits in einer früheren Runde aktiv waren, ignorieren empfangene Duplikate und nehmen erst beim Transport einer neuen Nachricht wieder teil. Die Anzahl der Runden für die Übertragung ist jedem Knoten bekannt und wird durch den maximalen Netzwerkdurchmesser festgelegt.

Zeitbedarf Sei d_{seqBB} die Zeit, um im *Cooperative Encoding* eine Sequenz aus k Bits mittels *Black Burst Frame* an die 1-Hop-Nachbarn zu versenden. Dann kann die Dauer einer Übertragung mittels *Cooperative Encoding* bei einer gewünschten Rundenanzahl *rounds* wie folgt berechnet werden:

$$d_{coop} = d_{seqBB} \cdot rounds$$

3.3 Algorithmus für den Routenaufbau

Der Algorithmus gliedert sich in vier Phasen, die nacheinander durchlaufen werden. Es soll eine Route zwischen den Knoten *src* und *dst* aufgebaut werden, die die QoS-Anforderungen *QoS* erfüllt. Um mehrere Routen zum selben Ziel zu unterstützen, wird für jede Route vom Initiator der Suchanfrage eine lokal eindeutige Id vergeben. Diese wird später in jedes Datenpaket integriert und ergibt zusammen mit der

Absenderadresse die global eindeutige Id der Route, auf der die Nachricht zum Ziel geleitet werden soll. Ein weiterer Vorteil ist, dass auf diese Art und Weise zwischen zwei Knoten mehrere Pfade unterschiedlicher Dienstgüte aufgebaut werden können.

3.3.1 Phase 1 - Route Request Selection Phase

Ziel von Phase 1 ist es, den Netzwerkteilnehmern die Anforderung eines Leitwegs mitzuteilen und die Anfragen zu sequenzialisieren.

Der Start von Phase 1 muss entweder dynamisch signalisiert werden oder am Anfang von speziell in MacZ dafür konfigurierten, allen Knoten bekannten Zeitslots liegen. So ist sichergestellt, dass alle Netzwerkknoten gleichzeitig Phase 1 betreten und sich auf das Empfangen bzw. Senden von Leitweganforderungen mittels *Arbitrating Encoding* einstellen.

Der Quellknoten *src* flutet das Netzwerk mit einem *RouteRequest*-Paket, welches seine eigene Adresse, die Adresse von *dst* und die kodierte QoS-Anforderung *QoS* enthält. Das *RouteRequest*-Paket wird mit kollisionsgeschützten *Black Burst Frames* unter Verwendung des *Arbitrating Encoding* verschickt, so dass sich von potentiell mehreren im Netz befindlichen, konkurrierenden Leitweganforderungen nur eine dominante durchsetzt. Dies ist die Anfrage, deren Initiator die numerisch höchste Adresse hat. Die Leitweganforderungen werden dadurch sequenzialisiert und die folgenden drei Phasen können ohne Störungen durch andere Vorgänge durchlaufen werden. Diese Form der Abarbeitung hat außerdem den Vorteil, dass Ressourcen nicht vorreserviert werden müssen, da es keine Konkurrenten gibt. Knoten, die nicht das Ziel der Route sind und ein *RouteRequest*-Paket erhalten, prüfen, falls dies schon möglich ist, ob sie genügend Ressourcen haben, um am Routing teilzunehmen. Wenn sie keine ausreichende Kapazität besitzen, ziehen sie sich vom aktuellen Suchvorgang zurück. Kann eine Entscheidung auf Grund der verwendeten Metrik erst später getroffen werden (z.B. Gesamtverzögerung entlang eines Pfades) oder ist ein Knoten in der Lage die Anforderungen zu erfüllen, geht er in Phase 2 und wartet auf ein *RouteReply*-Paket. Erhält der Zielknoten ein *RouteRequest*, wartet er, bis sich die Anforderung aus Phase 1 komplett im Netz ausgebreitet hat und initiiert dann Phase 2.

Die Dauer von Phase 1 ist vom maximalen Netzwerkdurchmesser bestimmt und kann von jedem Knoten, wie in Abschnitt 3.2.2.1 beschrieben, ermittelt werden. Der benötigte Parameter *maxDiam* für den maximalen Netzwerkdurchmesser ist in MacZ konfiguriert. Die Länge der Bit-Sequenz ist durch die feste Größe eines *RouteRequest* bestimmt.

Am Ende von Phase 1 weiß jeder Knoten, dass eine Route vom Ziel- zum Quellknoten mit gewissen QoS-Anforderungen benötigt wird. Dies gilt insbesonder für den Zielknoten. Nur eine von potentiell mehreren Anfragen setzt sich durch und wird in den folgenden drei Phasen weiter bearbeitet.

Pseudo-Code für Phase 1

```
// ALGORITHM DATA STRUCTURES
```

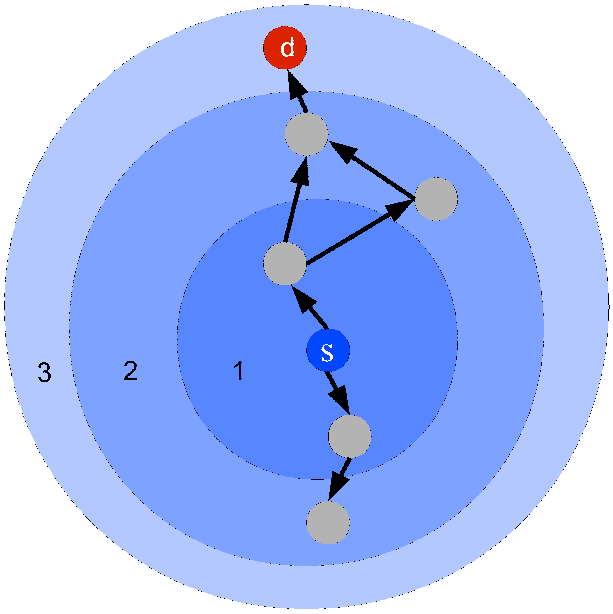

Abbildung 3.4: Phase 1 - Verbreitung eines Route Requests

```
// Route Requests structures are used to transmit route requests
// to a network node and to store information on the currently
// performed route search
struct RouteRequest {
    // id of the route's source node
    NodeId srcId;

    // id of the route's destination node
    NodeId dstId;

    // distance of the local node to the destination in hops
    int distance;

    // id of the successor in the route
    NodeId nextHop;

    // id of the predecessor in the route
    NodeId prevHop;

    // locally unique route id created by the source node of the route
    RouteId connId;

    // QoS requirements for the route
    QoSRequirements QoS;
}
```

```
// Route Reply structures are used to transmit route reply messages
struct RouteReply {
    // distance of the receiving node to the destination in hops
    int distance;

    // id of the route's destination node
    NodeId dstId;
}

// Construction Request structures are used to transmit construction
// request messages
struct ConstructionRequest {
    // id of the message's sender
    NodeId srcId;

    // hop distance to the destination a potential successor node needs
    // to have in order to participate in the routing
    int distance;

    // locally unique route id created by the source node of the route
    RouteId connId;
}

// Construction Reply structures are used to transmit construction
// reply messages
struct ConstructionReply {
    // id of the message's sender
    NodeId srcId;

    // id of its predecessor in the route
    NodeId dstId;
}


// IMPORTANT LOCAL VARIABLES

// phase the local node is in
NodeState state;

// Route Request from the service user
RouteRequest localRequest;

// Route Request received via network
RouteRequest envRequest;

// Route Request storing information about the current discovery process
RouteRequest routeRequest;
```

```
// Route Reply received via network
RouteReply envReply;

// Construction Request received via network
ConstructionRequest constrRequest;

// Construction Reply received via network
ConstructionReply constrReply;


// PHASE1: source -> destination, Route Request Selection Phase

// source node
if (state = idle && localRequest != undef)
    // service user requests a route
    routeRequest := localRequest
    state := sender.phase1
    distance(routeRequest) := 0
    nextHop(routeRequest) := undef
    sendRouteRequest(id(self), dstId(routeRequest), QoS(routeRequest))
    receiveTimer := new Timer
    exptime(receiveTimer) := receiver.timeout1
endif

// intermediate node
if (state = idle && envRequest != undef)
    routeRequest := envRequest
    if (dstId(routeRequest) = id(self))
        // node is the destination of the requested route => node starts
        // phase 2
        state := receiver.phase2
        distance(routeRequest) := 0
        nextHop(routeRequest) := undef
        // wait for completion of phase 1
        awaitPhase1Completion()
        sendRouteReply(dstId(routeRequest), distance(routeRequest) + 1)
        receiveTimer := new Timer
        exptime(receiveTimer) := receiver.timeout2
    else
        // node is neither source nor destination,
        state := intermediate.phase1
        routeRequest := envRequest
        receiveTimer := new Timer
        exptime(receiveTimer) := intermediate.timeout1
    endif
endif
```

3.3.2 Phase 2 - Route Discovery Phase

In Phase 2 wird mit der eigentlichen Suche nach einer Route begonnen. Vom Zielknoten ausgehend breiten sich *RouteReply*-Nachrichten wellenartig im Netz aus bis der Quellknoten erreicht wird oder eine Nachricht die maximal für eine Route erlaubte Anzahl von Knoten besucht hat.

Der Zielknoten sendet ein *RouteReply*-Paket mit seiner eigenen Adresse und einem auf den Wert 1 gesetzten Hop-Zähler. Dieser dient dazu, den Empfänger über seine minimale Distanz zum Zielknoten zu informieren. Empfängt ein Knoten, der sich in Phase 2 befindet, ein *RouteReply*-Paket, prüft er, ob er die geforderten QoS-Anforderungen erfüllen kann. Ist dies der Fall, merkt er sich den Wert des Hop-Zählers, erhöht den Zähler im Paket, sendet es weiter und geht in Phase 3. Ansonsten zieht er sich aus der aktuellen Routensuche zurück. Viele Pfadeigenschaften sind nicht an Hand von rein lokalen Informationen überprüfbar, so dass bei den meisten Metriken Netzwerk-Statusinformationen mit Nachbarknoten ausgetauscht werden müssen. Im Fall der Bandbreiten-Metrik sind dies Informationen über den Reservierungsstatus von Zeitslots bei den Nachbarn. Im Folgenden wird davon ausgegangen, dass der für Routing-Entscheidungen benötigte Netzwerkstatus lokal bekannt ist.

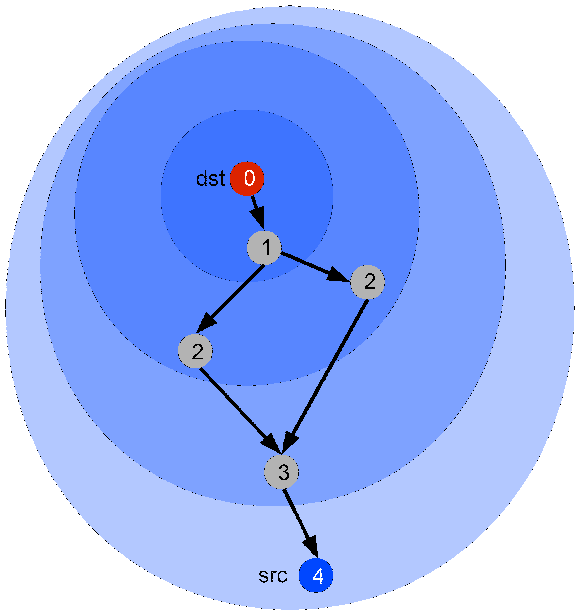

Abbildung 3.5: Phase 2 - Distanzen zum Zielknoten *dst*

Empfängt der Quellknoten ein *RouteReply*, dann ist das entsprechende Paket entlang einer *hop-minimalen* Route geschickt worden, die die QoS-Anforderungen erfüllt. Die minimale Hop-Distanz zum Zielknoten auf einer solchen Route entnimmt der Quellknoten dem Hop-Zähler des *RouteReply*. Phase 2 endet, wenn sich auf Grund des Netzwerkdurchmessers bzw. der maximalen Routenlänge keine *RouteReply*-Nachrichten mehr in Umlauf befinden können. Dieser Zeitpunkt markiert den Beginn von Phase 3 und ist allen Knoten bekannt. Er kann wie in Abschnitt 3.2.2.3 berechnet werden, wobei die gewünschte Rundenzahl dem maximalen Netzwerkdurchmesser und k der festen Länge eines *RouteReply* in Bits entspricht.

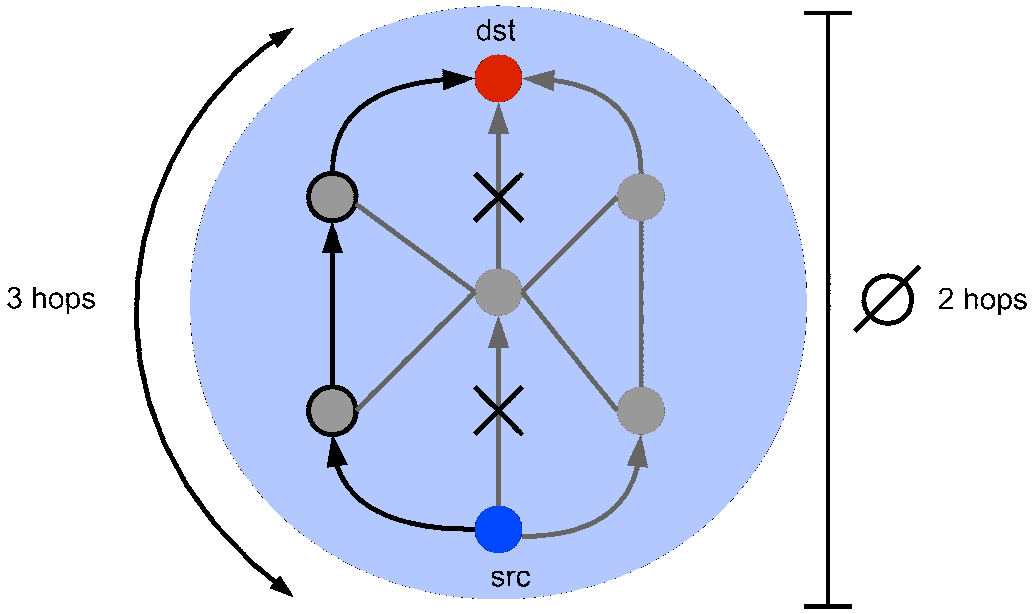

Abbildung 3.6: Routen auf dem Netzwerkrand

Am Ende von Phase 2 kennt jeder Knoten, der am Routing teilnehmen möchte und auf Grund seiner Ressourcen auch dazu in der Lage ist, seine minimale Hop-Distanz auf dem Weg zum Zielknoten. In Abbildung 3.5 sind für eine Beispieltopologie die Minimaldistanzen von Netzwerkteilnehmern zum Ziel dargestellt.

Unter Umständen existiert zwischen zwei Knoten keine "direkte" Verbindung, d.h. die Anzahl der Hops in einer Route mit den *gewünschten Eigenschaften* ist größer als die tatsächliche minimale Hop-Distanz zwischen ihnen. Unser Verfahren ist in solchen Situationen in der Lage, eine Route mit suboptimaler Hop-Anzahl aufzubauen. Die maximale Länge solcher Routen ist nicht vom Netzwerkdurchmesser, sondern durch einen Konfigurationsparameter bestimmt, der allen Netzwerkteilnehmern bekannt ist. In Abbildung 3.6 ist ein Pfad auf dem Netzwerkrand illustriert, der nicht die minimale Zahl von Knoten besitzt.

Pseudo-Code für Phase 2

```
// PHASE2: destination -> source, Route Discovery Phase

if (expired(receiveTimer))
    // something went wrong => free resources if necessary,
    // report routing failure to upper layer and become idle
    reportFailure()
    routeRequest := undef
    freeResources()
    state := idle
else
    // intermediate node
```

```
    if (state = intermediate.phase1 && envReply != undef)
        if (checkResources(QoS(routeRequest)))
            // node received a routing reply message and has enough
            // resources => it enters phase2
            // and now knows its distance from the destination node. The
            // reply is forwarded via cooperative encoding with
            // incremented hop distance field
            routeReply := envReply
            state := intermediate.phase2
            distance(routeRequest) := distance(routeReply)
            nextHop(routeRequest) := undef
            sendRouteReply(dstId(routeReply), distance(routeReply) + 1)
            receiveTimer := new Timer
            exptime(receiveTimer) := intermediate.timeout2
        else
            // not enough resources
            state := idle
            routeRequest := undef
        endif
    endif

    // destination node
    if (state = receiver.phase2 && envConstrRequest != undef)
        // destination node receives construction request and thus is
        // notified about the end of phase 3 => destination node starts
        // phase 4. From its perspective the route is now established.
        state := receiver.established
        constrRequest := envConstrRequest
        reserveResources()
        // wait for completion of phase 2
        awaitPhase2Completion()
        sendConstructionReply(id(self), srcId(constrRequest))
    endif
endif
```

3.3.3 Phase 3 - Route Selection Phase

Der Quellknoten sendet ein *ConstructionRequest*-Paket mit seiner eigenen Adresse, seiner um eins verringerten Distanz zum Zielknoten und einer lokal eindeutigen Verbindungs-Id an die direkten Nachbarn. Diese Id muss später neben der Quelladresse in jedes Datenpaket eingefügt werden. Empfängt ein Knoten in Phase 3 ein *ConstructionRequest*, prüft er zunächst, ob seine Distanz zum Ziel dem Hop-Counter im empfangenen Paket entspricht. Falls dies zutrifft, merkt er sich die Adresse des Absenders und die Verbindungs-Id, setzt die eigene Adresse und seine Distanz zum Ziel ein und sendet das Paket weiter. Der mitgeschickte Hop-Counter sorgt dafür, dass nur Knoten, die sich auf der kürzesten, machbaren Route befinden, wirklich in-

tegriert werden. Abbildung 3.7 illustriert die Auswahl von Netzwerkknoten für eine solche Route.

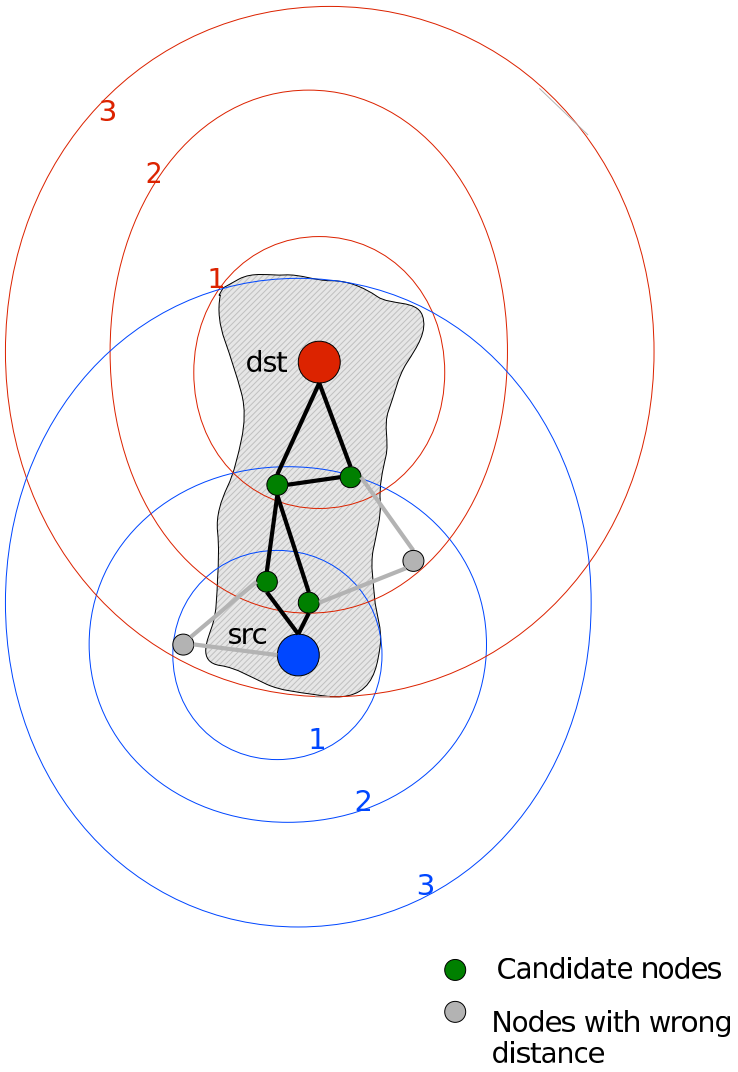

Abbildung 3.7: Phase 3 - Ermittlung von Nachfolgeknoten

In Phase 3 kann das *Hidden Station Problem* auftauchen. Wenn mehrere Knoten gleichzeitig ein *ConstructionRequest* weitersenden, kann es sein, dass es beim Empfänger zu Kollisionen der beiden Pakete kommt. In Phase 3 werden die Pakete allerdings mit kollisionsgeschützten *Black Burst Frames* versendet, die sich im geschilderten Fall zwar nicht gegenseitig zerstören, dafür allerdings auf dem Medium ODER-verknüpft werden. Hier verwendet der Algorithmus das *Restricted Arbitrating Encoding*, das dafür sorgt, dass sich im Fall mehrerer Vorgängerknoten exakt einer durchsetzt (siehe Abschnitt 3.2.2.1). Wenn der Zielknoten ein *ConstructionRequest* mit einem Hop-Counter von 0 empfängt, ist eine hop-minimale Route gefunden worden, die die Anforderungen erfüllt. Er merkt sich die Adresse des Absenders und wartet auf den Abschluss von Phase 3.

Sie endet, wenn das *ConstructionRequest* am Zielknoten angekommen ist. Da die Knoten ihre Distanz zum Ziel und die Dauer einer Übertragung mittels *Restricted*

Arbitrating Encoding kennen, ist es für sie jeweils möglich beim Empfang eines *ConstructionRequest* die verbleibende Zeit bis zum Ende von Phase 3 zu berechnen. Sie ergibt sich zu

$$d_{p3end} = d_{restrarb} \cdot maxDiam$$

$d_{restrarb}$ ist Dauer einer Übertragung von k Bits (feste Länge eines *ConstructionRequest*) mittels *Restricted Arbitrating Encoding*. $maxDiam$ ist der in MacZ konfigurierte Parameter für den maximalen Netzwerkdurchmesser. Für Details siehe Abschnitt 3.2.2.2.

Pseudo-Code für Phase 3

```
// PHASE3: source -> destination, Route Selection Phase

if (expired(receiveTimer))
    // something went wrong => free resources if necessary,
    // report routing failure to upper layer and become idle
    reportFailure()
    routeRequest := undef
    freeResources()
    state := idle
else
    // source node
    if (state = sender.phase1 && envRouteReply != undef)
            // on reception of a route reply the source node notices
            // completion of phase 2
            routeReply := envRouteReply
            state := sender.phase3
            distance(routeRequest) := distance(routeReply)
            nextHop(routeRequest) := undef
            receiveTimer := new Timer
            exptime(receiveTimer) := sender.timeout3

            // node generates a locally unique connection id and
            // initiates phase 3
            connId(constrRequest) := generateId()
            connId(routeRequest) := connId(constrRequest)
            distance(constrRequest) := distance(routeReply)
            // wait for completion of phase 2
            awaitPhase2Completion()
            sendConstructionRequest(id(self), distance(constrRequest) - 1,
                connId(constrRequest))
    endif

    // intermediate node
    if (state = intermediate.phase2 && envConstrRequest != undef &&
        distance(routeRequest) = distance(envConstrRequest))
```

```
        // node has received a construction request and has the demanded
        // distance from the source node => it enters phase 3 and forwards
        // construction request in direction of the destination node,
        // hop distance field is decremented
        constrRequest := envConstrRequest
        state := intermediate.phase3

        // the source node of the received construction request won the
        // competition due to his dominant id, this actually determines
        // the route. The other predecessors will timeout
        prevHop(routeRequest) := srcId(constrRequest)

        // copy connection id for later use in data packets
        connId(routeRequest) := connId(constrRequest)

        sendConstructionRequest(id(self), distance(constrRequest) - 1,
            connId(routeRequest))
        receiveTimer := new Timer
        exptime(receiveTimer) := intermediate.timeout3
    endif
endif
```

3.3.4 Phase 4 - Route Establishment Phase

Der Zielknoten sendet ein *ConstructionReply*-Paket mit seiner eigenen und der Adresse des Knotens, von dem er in Phase 3 den *ConstructionRequest* empfangen hat (Vorgänger mit der dominanten Adresse). Empfängt ein Knoten in Phase 4 ein *ConstructionReply*, dessen Zieladresse zu seiner eigenen identisch ist, weiß er, dass er zum Pfad gehört. Er reserviert die benötigten Ressourcen und speichert die Absenderadresse aus der *ConstructionReply* als Adresse des nächsten Knotens in Richtung des Ziels. Schließlich sendet er die *ConstructionReply* weiter. Sobald der Quellknoten ein *ConstructionReply* mit seiner Adresse als Ziel empfängt, merkt er sich die Adresse des Absenders. Die Route ist nun aufgebaut und bereit für den Datentransfer. Nach Phase 4 kennt jeder Knoten die Adresse des nächsten Knotens in Richtung Zielknoten. Außerdem sind auf allen beteiligten Knoten die benötigten Ressourcen reserviert worden, die Route ist aufgebaut und kann nun verwendet werden.

Pseudo-Code für Phase 4

```
// PHASE4: destination -> source, Route Establishment Phase
// route propagation

if (expired(receiveTimer))
    // something went wrong => free resources if necessary,
    // report routing failure to upper layer and become idle
    reportFailure()
```

```
    routeRequest := undef
    freeResources()
    state := idle
else
    // destination node
    if (state = receiver.phase2 && envConstrRequest != undef)
        // destination node receives construction request and thus is
        // notified about the end of phase 3 => destination node starts
        // phase 4
        state := established
        constrRequest := envConstrRequest
        reserveResources()
        // wait for completion of phase 3
        awaitPhase3Completion()
        sendConstructionReply(id(self), srcId(constrRequest))
    endif

    // intermediate node
    if (state = intermediate.phase3 && envConstrReply != undef)
        if (srcId(envConstrReply) = id(self))
            // node receives construction reply with its id as destination
            // => it now knows that it's part of the route and resources
            // have to be reserved. Construction reply is forwarded in
            // direction of the source
            constrReply := envConstrReply
            state := established
            nextHop(routeRequest) := srcId(constrReply)
            reserveResources()
            sendConstructionReply(id(self), prevHop(routeRequest))
        else
            // node is not part of the route, a different id dominated
            //in phase 3
            receiveTimer := undef
            state := idle
        endif
    endif

    // source node
    if (state = sender.phase3 && envConstrReply != undef)
        // node receives construction reply => the route is now
        // established and ready for data transfer
        constrReply := envConstrReply
        state := established
        nextHop(routeRequest) := srcId(constrReply)
        reserveResources()
    endif
endif
```

3.3.5 Paketformate

Route Request *RouteRequest(src, dst, qos)*

src Adresse des Quellknoten, der eine Route benötigt (10 Bits)

dst Adresse des Zielknotens der Route (10 Bits)

qos Quality-of-Service-Anforderungen an die Route (5 Bits)

Die Angabe des Quellknotens entscheidet beim Versenden einer Anfrage mittels *Arbitrating Encoding*, welcher von möglicherweise mehreren Konkurrenten sich schließlich durchsetzt.

Route Reply *RouteReply(dst, hops)*

dst Adresse des Zielknotens (10 Bits)

hops Distanz zum Zielknoten in Hops (12 Bits)

Die Adresse des Zielknotens wird von jedem Knoten in Phase 2 unverändert im Paket weitergeleitet. Diese Information ist zu diesem Zeitpunkt jedoch bereits auf allen Knoten vorhanden und ihre erneute Verbreitung kann in späteren Verfeinerungen des Algorithmus weggelassen werden. Ein Knoten, der eine Route Reply empfängt, kann dem *hops*-Zähler seine minimale Distanz zum Zielknoten entnehmen. Diese bezieht sich jedoch nur auf Routen, die die QoS-Anforderungen erfüllen.

Construction Request *ConstructionRequest(src, hops, rId)*

src Adresse des Paket-Absenders (10 Bits)

hops Distanz zum Zielknoten in Hops (12 Bits)

rId Routen-Id (10 Bits)

Beim Versenden eines *Construction Requests* fügt der Absender seine eigene Adresse in das *src*-Feld ein. Über das *src*-Feld wird in Phase 3 mittels *Arbitrating Encoding* entschieden, welcher Sendeknoten sich durchsetzt. Der Empfänger merkt sich diesen Gewinner als Vorgängerknoten in der berechneten Route. An Hand der *hops*-Zahl entscheidet ein Empfänger, ob er als Knoten auf der hop-minimalen Route (mit gültigen QoS-Garantien) zum Zielknoten in Frage kommt. Dies ist der Fall, wenn seine Entfernung zum Zielknoten mit der *hops*-Zahl im empfangenen Paket übereinstimmt. *rId* ist die vom Quellknoten vergebene, lokal eindeutige Id für diese Route.

Construction Reply *ConstructionReply(src, prevHop)*

src Adresse des Paket-Absenders (10 Bits)

prevHop Adresse des Knotens, der als Vorgängerknoten in der Route fungieren soll (10 Bits)

Ein Empfänger, der eine *Construction Reply* mit seiner Adresse als *prevHop* erhält, weiß, dass er nun Teil der Route ist. Die im Paket enthaltene *src*-Adresse ist der nächste Knoten der Route auf dem Weg zum Zielknoten.

Datenpakete *DataReq(src, rId, data)*

src Adresse des Quellknotens der Route (10 Bits)

rId Vom Quellknoten festgelegte Routen-Id (10 Bits)

data Zu übermittelnde Daten (variable Anzahl Bits)

Sobald eine Route zwischen Quell- und Zielknoten aufgebaut ist, können Daten zwischen beiden ausgetauscht werden. Nun ist es jedoch möglich, dass zwischen den selben beiden Knoten mehrere Routen bestehen. Ist nun ein Knoten x Teil von zwei solchen Routen, kann er nicht allein an Hand der Information, wer Sender und wer Empfänger ist, entscheiden auf welchem Weg ein Paket weitergeleitet werden soll (Annahme: die Nachfolgeknoten auf den zur Wahl stehenden Routen sind verschieden). Um verschiedene Routen vom selben Quellknoten aus unterscheiden zu können, wird eine Routen-Id *rId* eingeführt. Zusammen mit der Adresse des Quellknotens bildet diese eine netzwerkweite eindeutige Id. Der Quellknoten muss allerdings dafür sorgen, dass jeder von ihm initiierte Routing-Vorgang eine *lokal* eindeutige Id erhält. Beteiligte Knoten müssen sich für jede Route den Quellknoten und die dazugehörige Routen-Id merken, um erhaltene Pakete korrekt weiterleiten zu können. Falls zusätzlich die Id des Zielknotens mit in die Routing-Informationen integriert wird, muss diese in Datenpaketen nicht mehr mitgesendet werden. Es genügt lediglich die Id des Quellknotens und die Routen-Id einzufügen.

Kapitel 4

SDL-System

In diesem Kapitel wird die SDL-Spezifikation unseres Routing-Algorithmus erläutert (zu Informationen über SDL siehe [ITU99]). Die Spezifikation wurde mit der Entwicklungsumgebung *Telelogic Tau 6.0* erstellt (siehe [tau]).

4.1 Architektur

Das BBQRt ist in zwei Blöcke aufgeteilt (siehe Abbildung 4.1). Der *BBQRt_Main*-Block enthält den eigentlichen Algorithmus. Er ist über einen Kanal mit der darüberliegenden Protokollschicht verbunden und bietet auf diesem Weg ein Interface für die Route-Discovery und den Datentransport an. Der *BBQRtCoDex* ist nach dem *CoDex*-Pattern entworfen (siehe dazu [GGR97]) und vermittelt zwischen der *MacZ*-Schicht und dem *BBQRt_Main*-Block. Hier werden Nachrichten aus der darüberliegenden Routing-Schicht für den Versand mittels *Black Burst Frames* kodiert und die Sendemodi von MacZ entsprechend den zu übertragenden Nachrichtentypen aktiviert.

4.1.1 Routing-Tabelle

Die Routing-Tabelle enthält für jeden möglichen Netzwerkknoten eine Liste. Dort sind alle von diesem Knoten ausgehenden, reservierten Routen verzeichnet, an denen der lokale Knoten beteiligt ist. Die Listeneinträge (Datentyp *RtEntry*) sind jeweils mit der vom Quellknoten vergebenen, lokal eindeutigen Routen-Id versehen. Die Größe der Routing-Tabelle wird durch eine konfigurierbare, maximale Anzahl an Verbindungen je Quellknoten begrenzt.

Wird eine aufgebaute Route längere Zeit nicht verwendet, sollen die zugehörigen Ressourcen wieder freigegeben werden. Um dies zu gewährleisten, wird jedem Routing-Tabelleneintrag ein SDL-Timer zugewiesen. Als Index für den Timer werden die Routen-Id und die Adresse des Quellknotens verwendet, die zusammen eine eindeutige Id für den Eintrag ergeben.

RtEntry (Eintrag in der Routing-Tabelle)

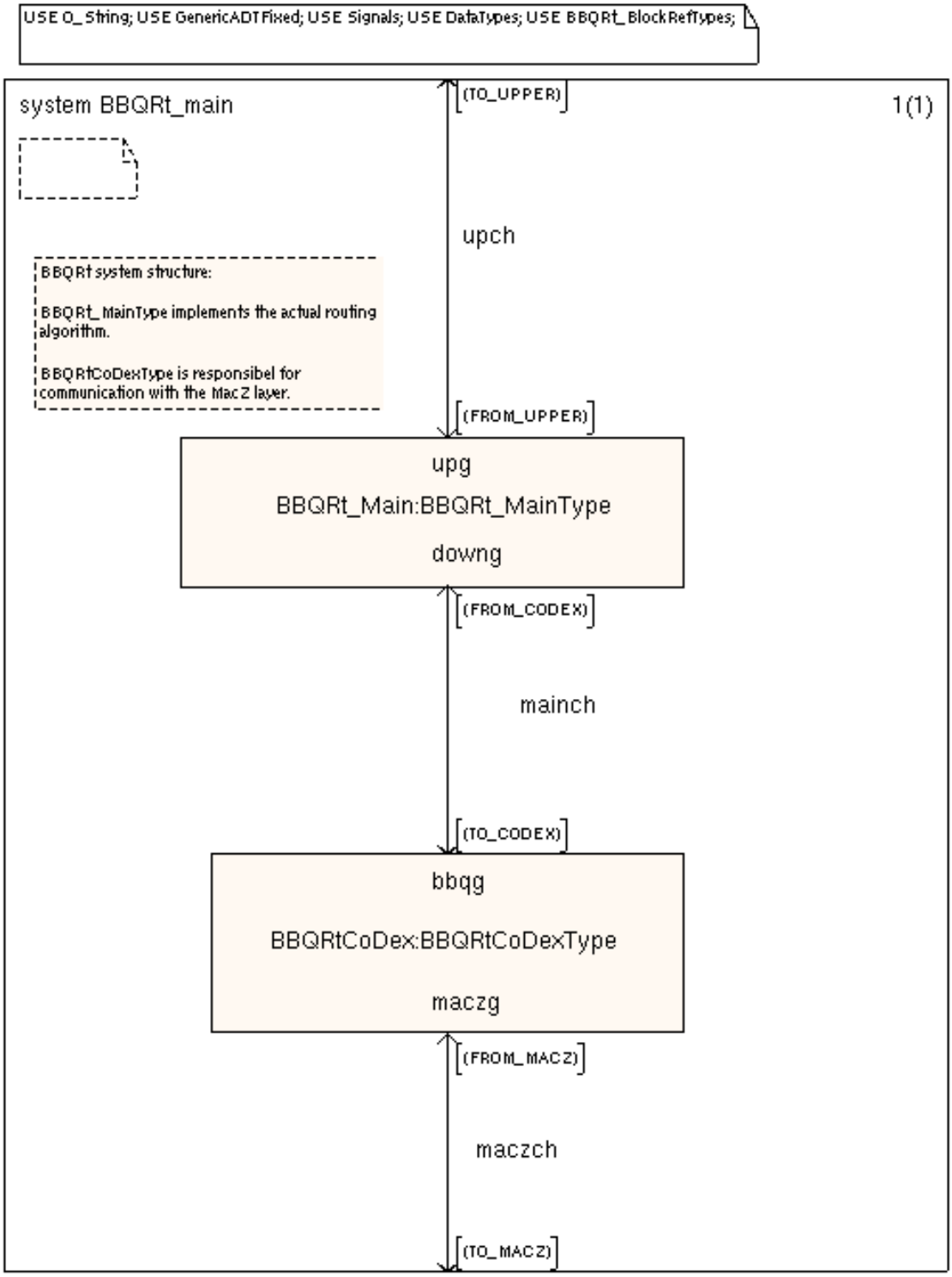

Abbildung 4.1: SDL-System

src : Address Adresse des Quellknoten der Route

dst : Address Adresse des Zielknotens der Route

prevHop : Address Adresse des Vorgängerknotens in der Route

nextHop : Address Adresse des Nachfolgeknotens in der Route

hops : Integer Distanz des lokalen Knotens zum Ziel der Route

qos : QoSReq Quality-of-Service-Anforderungen an die Route

rId : RouteId Vom Quellknoten vergebene, lokal eindeutige Routen-Id

valid : Boolean Zeigt an, ob diese Route gültig ist

RTimer(src : Address, rId : RouteId) (Timer für Tabelleneinträge)

src Adresse des Quellknotens der Route

rId Vom Quellknoten vergebene, lokal eindeutige Routen-Id

4.2 SDL-Signale

BBQ_RReq(src : Address, dst : Address, qos : QoSReq)

src Adresse des Knotens, der die Suche initiiert hat

dst Adresse des Zielknotens, zu dem ein Pfad benötigt wird

qos Dienstgüteanforderungen für die Route

Entspricht einer *RouteRequest*-Nachricht.

BBQ_RRep(dst : Address, hops : Integer)

dst Adresse des Zielknotens

hops Distanz des Empfängerknotens zum Zielknoten in Hops

Entspricht einem *RouteReply*-Paket.

BBQ_CReq(src : Address, hops : Address, rId : RouteId)

src Adresse des Paketversenders

hops Distanz zum Ziel, die der empfangende Knoten haben muss, um als Kandidat für die Route in Frage zu kommen

rId Lokal eindeutige Id, die der Initiator des Suchvorgangs für die Route festgelegt hat

Entspricht einer *ConstructionRequest*-Nachricht.

BBQ_CRep(src : Address, dst : Address)

src Adresse des Paketversenders

dst Adresse des Vorgängerknotens in der Route

Entspricht einer *ConstructionReply*-Nachricht.

BBQ_DataReq(src : Address, nextHop : Address, rId : RouteId, data : O_String)

src Adresse des Quellknotens der verwendeten Route

nextHop Adresse des nächsten Knotens in der Route

rId Lokal eindeutige Routen-Id

data Zu versendende Daten

Wird von der BBQRt-Schicht an den BBQRtCoDex gesendet, um Daten vom lokalen Knoten an den Zielknoten der Route zu übertragen.

BBQ_DataInd(src : Address, rId : RouteId, data : O_String)

src Quellknoten der Route

rId Routen-Id

data Empfangene Daten

Zeigt der BBQRt-Schicht an, dass über die Route mit dem Quellknoten *src* und der Routen-Id *rId* Daten empfangen wurden. Dieses Signal wird vom BBQRtCoDex verschickt.

BBQ_RouteId(rId : RouteId)

rId Lokal eindeutige Routen-Id

Die über BBQRt liegende Schicht des Initiators erhält über dieses Signal die für eine Route festgelegte Id. Alle anderen beteiligten Knoten erfahren die Id in Phase drei über *CReq(src, hops, rId)*-Nachrichten.

BBQ_IllegalRoute(rId : RouteId)

rId Lokal eindeutige Routen-Id

Signalisiert der Schicht über *BBQRt*, dass die in einem *APP_DataReq()*-Signal verwendete Routen-Id ungültig ist.

BBQ_RouteFailure(src : Address, rId : RouteId)

src Quellknoten der Route

rId Vom Quellknoten vergeben Routen-Id

Signalisiert der Schicht über *BBQRt*, dass die vom Quellknoten *src* ausgehende Route mit der Routen-Id *rId* nicht mehr verwendet werden kann.

APP_RReq(dst : Address, qos : Address)

dst Zielknoten der Route

qos Dienstgüteanforderungen an die Route

Wird vom Dienstnutzer verwendet, um eine Suchanfrage für eine Route vom lokalen Knoten zum Knoten *dst* mit den QoS-Anforderungen *qos* zu stellen.

APP_DataReq(rId : RouteId, data : O_String)

rId Lokal eindeutige Routen-Id

data Zu versendende Daten

Wird von der über BBQRt liegenden Schicht verwendet, um Daten auf einer bereits reservierten Route zu versenden. Zusammen mit der Adresse des Quellknotens ergibt die Routen-Id eine global eindeutige Id.

APP_DataInd(src : Address, rId : RouteId, data : O_String)

src Quellknoten der verwendeten Route

rId Vom Quellknoten vergebene Routen-Id

data Empfangene Daten

Wird von *BBQRt* verwendet, um Daten von der Route mit dem Quellknoten *src* und der Routen-Id *rId* an den Dienstnutzer weiterzuleiten.

APP_RouteEstablished(rId : RouteId)

rId Lokal eindeutige Routen-Id

Wird von *BBQRt* verwendet, um dem Dienstnutzer mitzuteilen, dass die angefragte Route mit der Id *rId* ab jetzt gültig und bereit für den Datentransfer ist.

4.3 Erwünschtes Systemverhalten (MSCs)

4.3.1 Initiierung eines Suchvorgangs

In Abbildung 4.2 sendet der Dienstnutzer *App* eine Anfrage für eine Route zum Zielknoten *dst* an die BBQRt-Schicht. BBQRt bestätigt die Anfrage mit einem *BBQ_RouteId*-Signal, das eine für diese Route generierte, lokal eindeutige Routen-Id enthält. BBQRt befindet sich nun in Phase 1 und sendet eine *RouteRequest*-Nachricht (*BBQ_RReq*-Signal) zum BBQRtCoDex. Dieser kodiert die Nachricht in eine Bit-Sequenz und reicht sie für die Übertragung an die MacZ-Schicht weiter. Auf der Empfängerseite ist zu sehen, wie der ankommende *Black Burst Frame* vom BBQRtCoDex in eine *RouteRequest*-Nachricht (*BBQ_RReq*-Signal) umgewandelt und an BBQRt weitergeleitet wird.

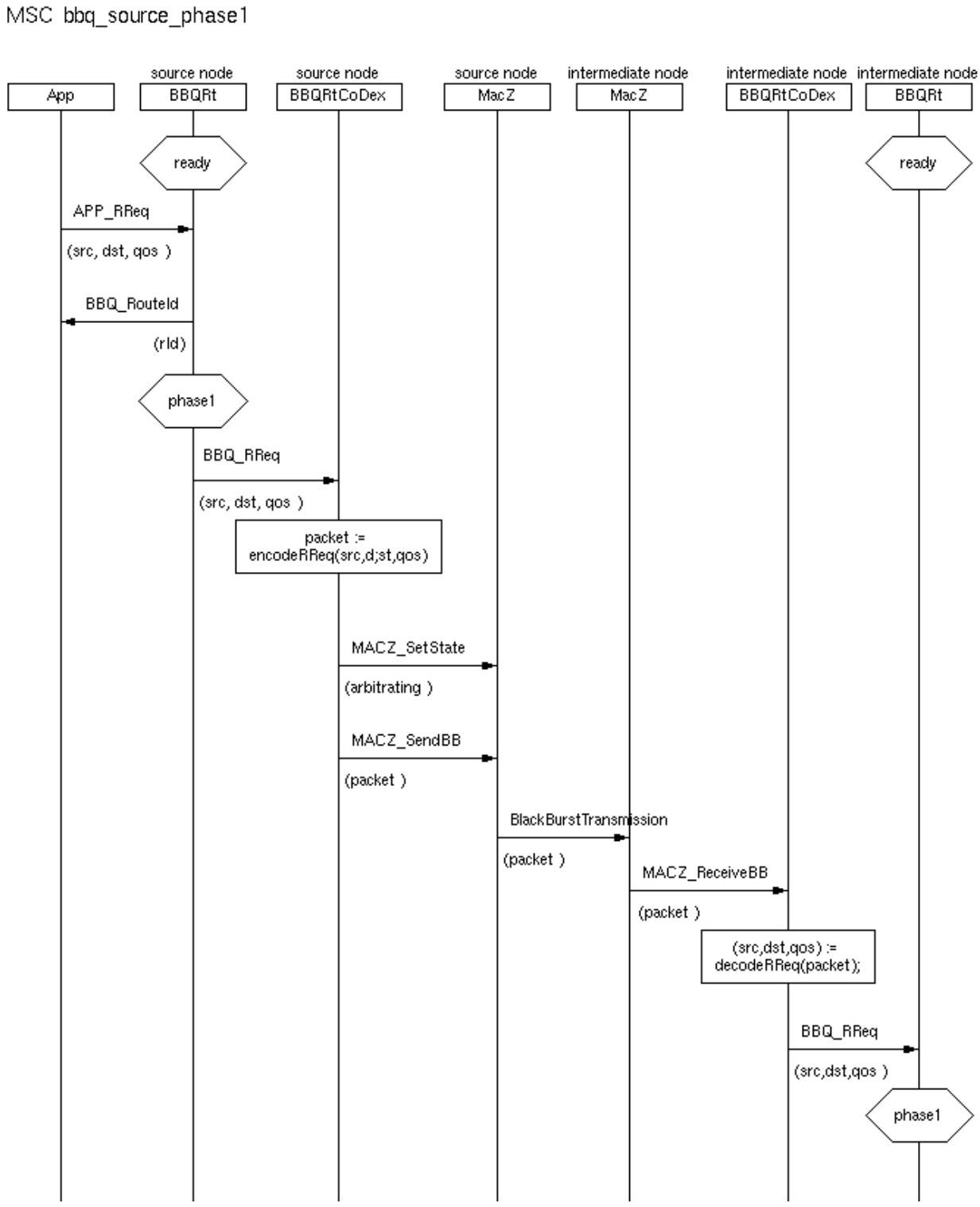

Abbildung 4.2: Initiierung eines Suchvorgangs

4.3.2 Ankunft beim Zielknoten

In Abbildung 4.3 ist zu sehen, wie der BBQRtCoDex des Zielknotens eine *RouteRequest*-Nachricht dekodiert und an BBQRt weiterreicht. Der Zielknoten wartet auf das Ende von Phase 1 und initiiert dann Phase 2, in dem seine BBQRt-Schicht eine *RouteReply*-Nachricht an den BBQRtCoDex sendet. Es folgt die Kodierung der Nachricht und die Weiterleitung an MacZ. Die MacZ-Schicht eines Zwischenknotens empfängt den gesendeten *Black Burst Frame* und schickt die Daten an den BBQRtCoDex des lokalen Knotens. Hier wird der Frame dekodiert und als *RouteReply*-Nachricht an BBQRt weitergegeben. Der Zwischenknoten befindet sich nun in Phase 2.

4.3.3 Beginn von Phase 3

In Abbildung 4.4 ist zu sehen, wie die *RouteReply*-Nachricht auf der kürzesten Route mit den gewünschten Eigenschaften den Quellknoten erreicht. Dieser schickt via BBQRtCoDex eine *ConstructionRequest*-Nachricht an seine Nachbarn und startet nach Beendigung von Phase 2 die Phase 3. Ein Zwischenknoten empfängt die Nachricht in Form eines *Black Burst Frames*, dekodiert sie und liefert sie schließlich als *ConstructionRequest*-Nachricht bei der BBQRt-Schicht ab. BBQRt merkt sich den Absenderknoten als Vorgänger in der Route und ist nun in Phase 3.

4.3.4 Abschluss des Suchvorgangs

In Abbildung 4.5 ist der erfolgreiche Abschluss einer Pfadsuche illustriert. Der Zielknoten erhält über MacZ und den BBQRtCoDex eine *ConstructionRequest*-Nachricht, wartet auf das Ende von Phase 3 und beginnt Phase 4. Er leitet eine *ConstructionReply*-Nachricht an den BBQRtCoDex weiter. Sie hat als Ziel die Adresse des Vorgängerknotens in der Route und wird im Gegensatz zu den anderen BBQRt-Nachrichten mittels *regulärer* MacZ-Frames und nicht mit *Black Bursts* übertragen. Beim Vorgängerknoten wird die Nachricht empfangen, vom BBQRtCoDex dekodiert und als *ConstructionReply*-Nachricht abgeliefert. Die Adresse des Absenders wird als Nachfolger in der Route gespeichert. Dann tritt der Empfänger in Phase 4 ein. Mit dem Abschluss von Phase 4 auf allen beteiligten Netzwerkknoten, ist die Route aufgebaut und kann für die Datenübertragung genutzt werden.

4.3.5 Weiterleitung von Datenpaketen

In Abbildung 4.6 ist zu sehen, wie ein Datenpaket auf einer gültigen Route bei einem Zwischenknoten ankommt. Dessen MacZ-Schicht sendet das empfangene Paket weiter an den BBQRtCoDex, wo es dekodiert und als *BBQ_DataInd*-Signal an BBQRt weitergeleitet wird. BBQRt stellt fest, dass der lokale Knoten nicht der Zielknoten ist und ermittelt an Hand der Routing-Tabelle den Nachfolgeknoten in der Route. Der BBQRtCoDex setzt im empfangenen Datenpaket als Zieladresse die des Nachfolgerknotens ein und gibt es zur Übertragung an MacZ weiter. MacZ schickt das Paket mittels regulärem Frame an den nächsten Knoten.

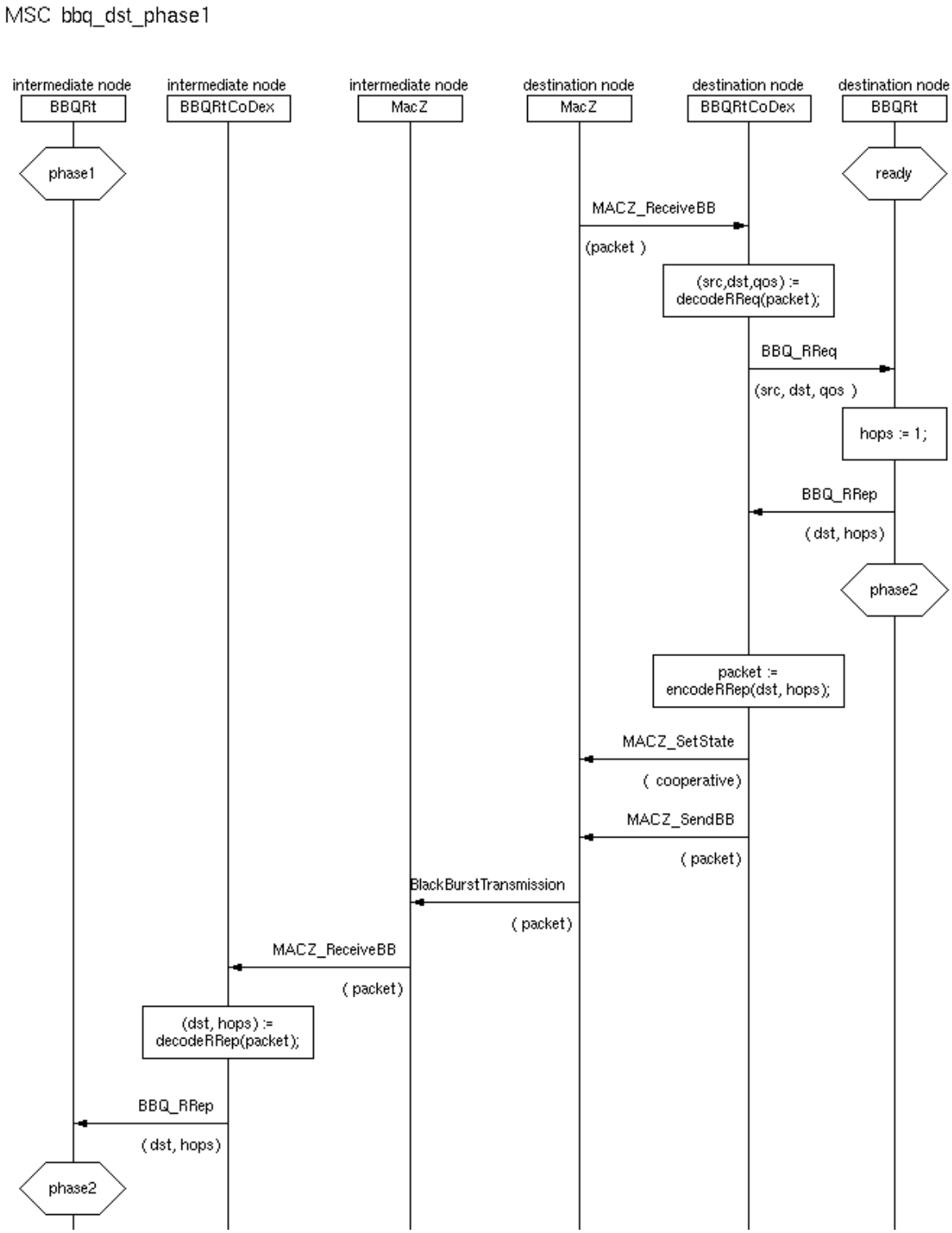

Abbildung 4.3: Ankunft einer *RouteRequest*-Nachricht beim Zielknoten

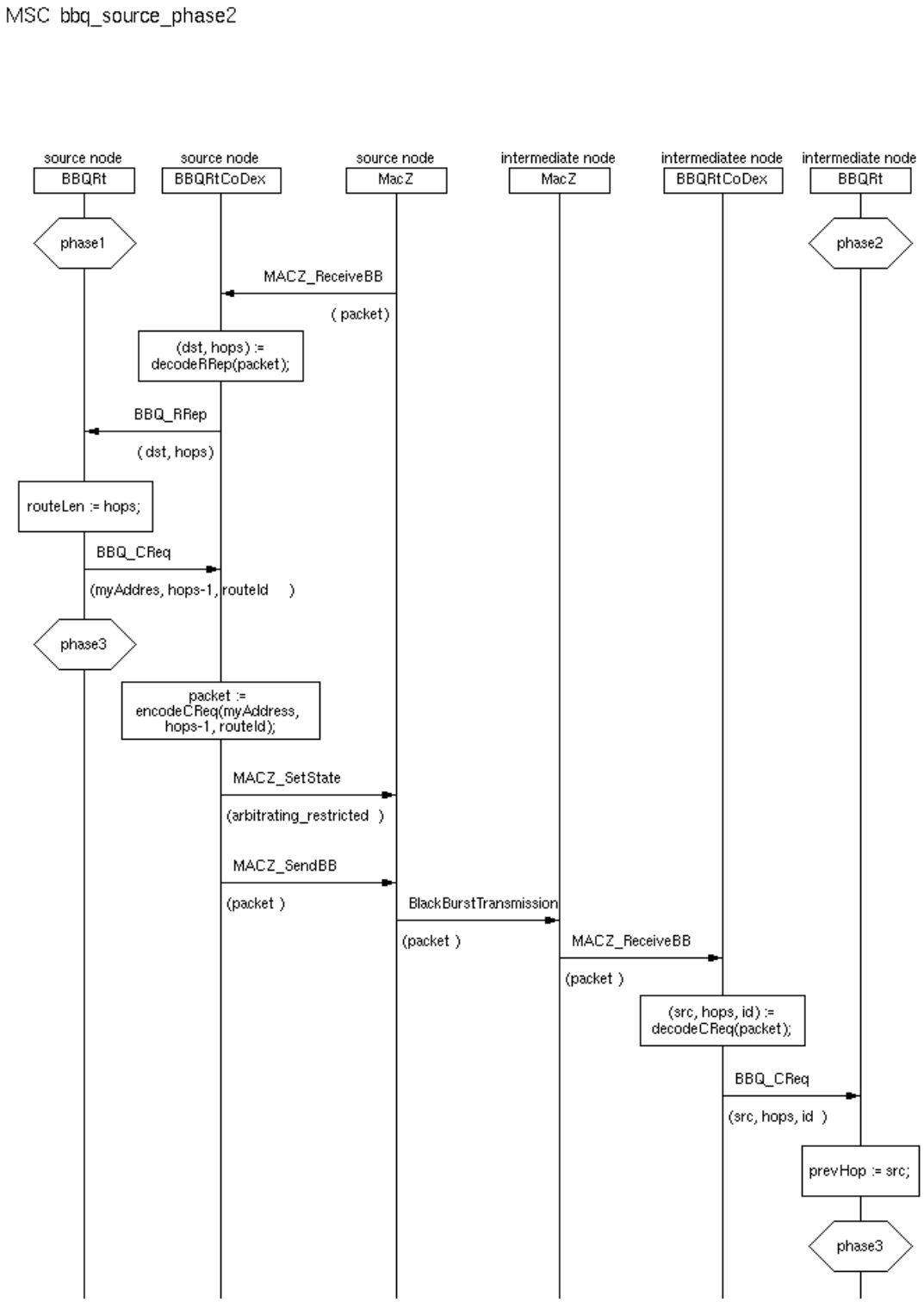

Abbildung 4.4: Beginn von Phase 3

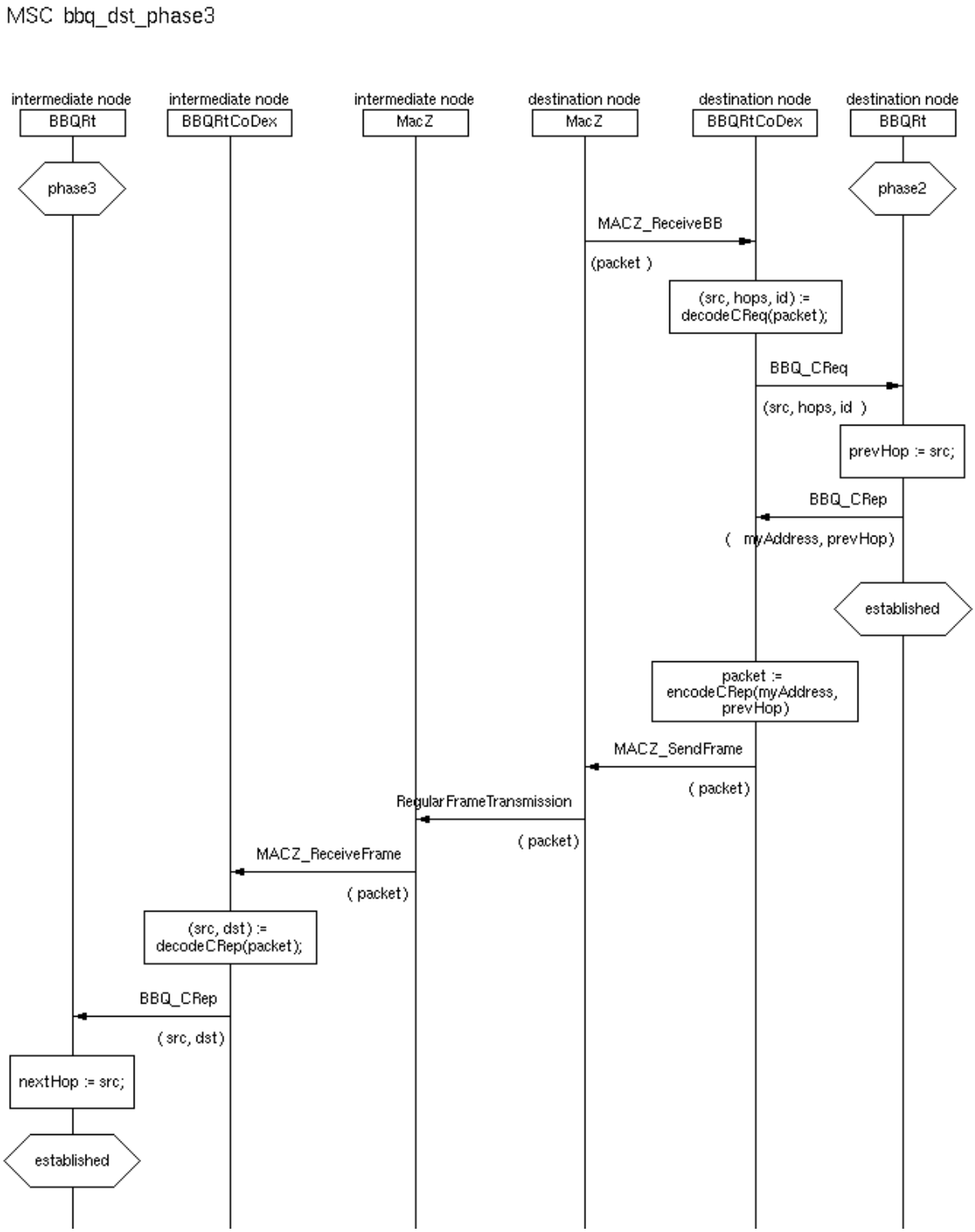

Abbildung 4.5: Abschluss des Suchvorgangs

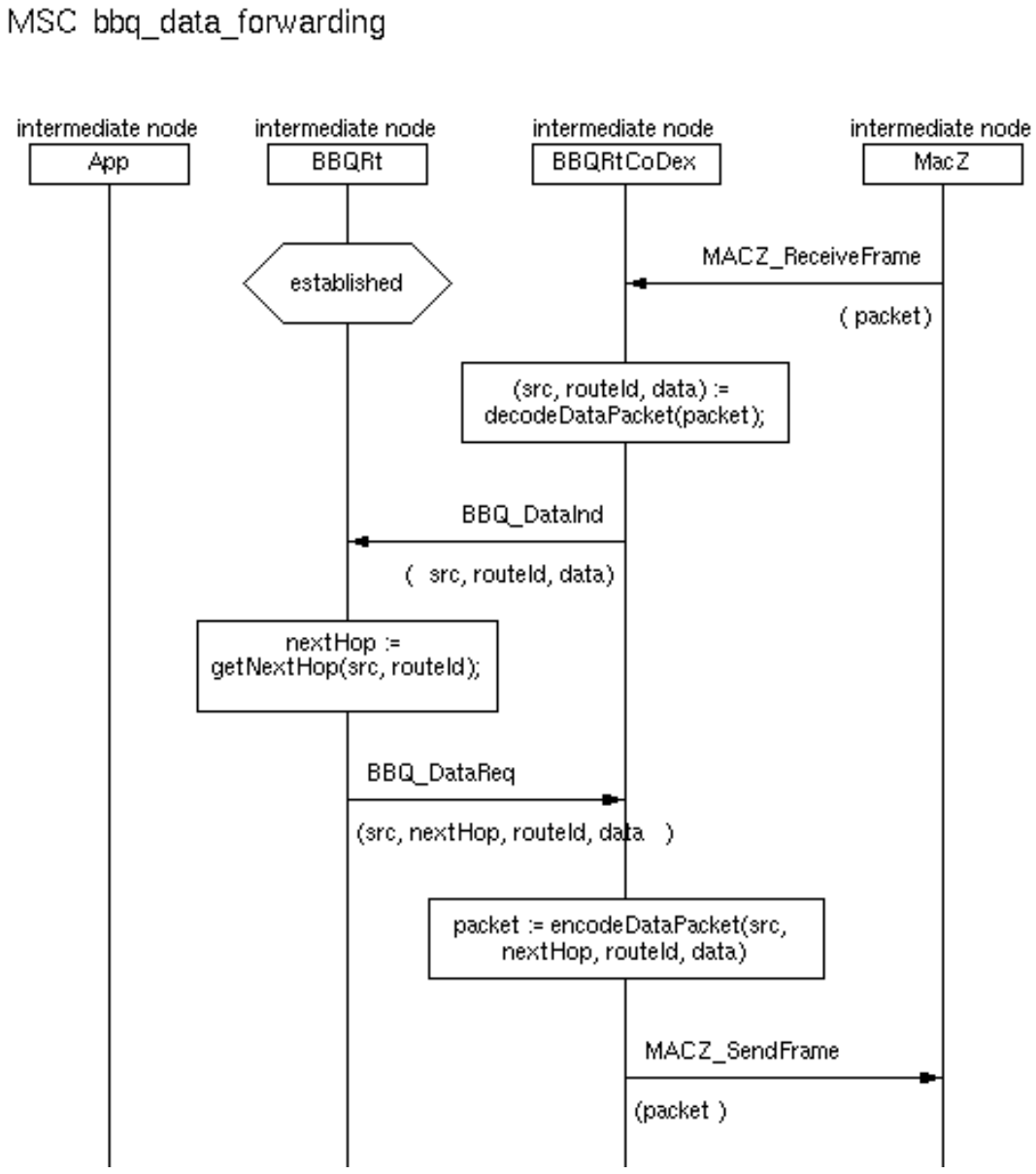

Abbildung 4.6: Paketweiterleitung

Kapitel 5

Zusammenfassung und Ausblick

Im Schlusskapitel möchten wir die Ergebnisse dieser Arbeit kurz zusammenfassen und einen Ausblick auf zukünftige Arbeiten geben.

In der vorliegenden Arbeit wurde ein neuartiges QoS-Routing-Verfahren für MANETs beschrieben, das besonders auf Szenarien mit hoher Knotendichte ausgelegt ist. Die vier Phasen des Algorithmus, die verwendeten Nachrichtenformate und ein Mechanismus zur Kodierung von QoS-Anforderungen wurden im Detail erklärt. Abschließend wurde das Verfahren in SDL spezifiziert.

Der Einsatz von kollisionsgeschützten *Black Burst Frames* und die Sequenzialisierung der Routensuche in unserem Algorithmus tragen maßgeblich zur Vermeidung wiederholten Sendens von Kontrollpaketen und somit zur Reduktion der Nachrichtenkomplexität bei. Außerdem ermöglicht die vom Netzwerkdurchmesser abhängige, deterministische Grenze für die Dauer eines Suchvorgangs ein effektives Energiemanagement. Knoten, die sich an der Routensuche beteiligen, wissen, in welchen Zeiträumen Datenverkehr zu erwarten ist und können in der restlichen Zeit auf überflüssiges Überwachen des Funkmediums verzichten.

Bis zum Abschluss dieser Arbeit lag keine Version der MacZ-Schicht vor, die die notwendigen Black-Burst-Encodings unterstützt, weshalb funktionale Tests nur innerhalb gewisser Grenzen möglich waren. In zukünftigen Arbeiten möchten wir das Black Burst Quality-of-Service Routing zunächst innerhalb von Simulationen mit *ns+SDL* (siehe [KGGR05]) und danach auf realer Hardware testen. Zusätzlich soll eine Evaluation der Effizienz von BBQRt in dichten Netzwerken und der Vergleich mit bereits existierenden Verfahren erfolgen.

Im Bereich der Metriken ist zu klären, wie der Aufwand steigt, wenn Informationen zwischen Netzwerkknoten ausgetauscht werden müssen, um unterschiedliche Metriken während der Pfadsuche berechnen zu können. Von Vorteil wäre hier ein bezüglich der Nachrichtenkomplexität effizienter Mechanismus zur Unterstützung *beliebiger* Metriken.

Anhang A

SDL-System auf CD

Literaturverzeichnis

[BGK07] Becker, Philipp, Reinhard Gotzhein und Thomas Kuhn: *MacZ - A Quality-of-Service MAC Layer for Ad-hoc Networks.* In: *Proceedings of 7th Int. Conference on Hybrid Intelligent Systems, Kaiserslautern, Germany*, Seiten 277–282, 2007.

[GGR97] Geppert, Birgit, Reinhard Gotzhein und Frank Rössler: *Configuring communication protocols using SDL patterns.* In: *Proceedings of 8th Int. SDL Forum, Evry, France*, Seiten 523–538, 1997.

[hLcT02] Liao, Wen hwa und Yu chee Tseng: *A TDMA-based bandwidth reservation protocol for QoS routing in a wireless mobile ad hoc network.* Communications, ICC 2002. IEEE International Conference on, 5:2002, 2002.

[ITU99] ITU-T Recommendation Z.100 (11/99): *Specification and Description Language (SDL).* International Telecommunication Union (ITU), 1999.

[JW04] Jawhar, Imad und Jie Wu: *A Race-Free Bandwidth Reservation Protocol for QoS Routing in Mobile Ad Hoc Networks.* In: *Proceedings of 37th Hawaii Int. Conference on System Science*, Band 9, 2004.

[KdI07] Kuhn, Thomas und José Irigon de Irigon: *An experimental evaluation of black burst transmissions.* In: *Proceedings of 5th ACM Int. Workshop on Mobility Management and Wireless Access, Chania, Crete Island, Greece*, Seiten 163–167, 2007.

[KGGR05] Kuhn, Thomas, Alexander Geraldy, Reinhard Gotzhein und Florian Rothländer: *ns+SDL - The Network Simulator for SDL Systems.* In: *Proceedings of 12th Int. SDL Forum, Grimstad, Norway*, Seiten 103–116, 2005.

[Lin01] Lin, Chunhung Richard: *An On-demand QoS Routing Protocol for Mobile Ad Hoc Networks.* In: *Proceedings of 20th Annual Joint Conference of the IEEE Computer and Communications Societies*, Seiten 1735–1744, 2001.

[tau] *Telelogic Tau SDL Suite – Homepage.* `http://www.telelogic.com/products/tau/sdl/index.cfm`.