COPPE/UFRJ

UM PROTOCOLO DE TRANSPORTE COLABORATIVO PARA REDES DE SENSORES SEM FIO

Eugênia Cristina Müller Giancoli Jabour

Tese de Doutorado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Doutor em Engenharia Elétrica.

Orientador: Aloysio de Castro Pinto Pedroza

Rio de Janeiro

Março de 2009

UM PROTOCOLO DE TRANSPORTE COLABORATIVO PARA REDES DE SENSORES SEM FIO

Eugênia Cristina Müller Giancoli Jabour

TESE SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Aprovada por:

Prof. Aloysio de Castro Pinto Pedroza, Dr.

Prof. Luiz Satoru Ochi, D.Sc.

Prof. Marcelo Gonçalves Rubinstein, D.Sc.

Profa. Maria Clicia Stelling de Castro, D.Sc.

Prof. Otto Carlos Muniz Bandeira Duarte, Dr.Ing.

RIO DE JANEIRO, RJ – BRASIL

MARÇO DE 2009

Jabour, Eugênia Cristina Müller Giancoli

UM PROTOCOLO DE TRANSPORTE COLABORATIVO PARA REDES DE SENSORES SEM FIO/Eugênia Cristina Müller Giancoli Jabour. – Rio de Janeiro: UFRJ/COPPE, 2009.

XV, 139 p.: il.; 29, 7cm.

Orientador: Aloysio de Castro Pinto Pedroza

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2009.

Referências Bibliográficas: p. 104 – 113.

1. Redes de sensores sem fio. 2. Protocolos de transporte. 3. Entrega confiável. I. Pedroza, Aloysio de Castro Pinto. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

Aos meus filhos: Eduardo,

Letícia, Fábio e Débora.

Agradecimentos

Para chegar até aqui passei por um grande caminho. Minha vida mudou. Reconsiderei verdades. Me cansei várias vezes. Perdi minha avó. Ganhei uma filha. Mudei de emprego. Mas sempre tive a certeza de que aqui eu queria chegar.

Muitos estavam ao meu redor durante a caminhada, contudo, alguns não puderam ficar até o fim. A todos eles, presentes ou ausentes, eu gostaria de agradecer...

Ao Filippe Jabour, pelo companheirismo e colaboração técnica.

Aos meus pais e irmã, por todo o amor e apoio ao longo da minha vida.

Ao William e Sarah, pela presença constante e pela ajuda nos momentos difíceis.

Ao Irmão Alcides Leopoldo Félix e ao José Ventura, pela eterna amizade.

Ao Padre Jan Zbigniew Czujak, pelo grande incentivo inicial.

Ao Aloysio, porque além de toda a sua orientação técnica, ele me ensinou um pouco mais sobre amizade, confiança e determinação. Ele tornou meu caminho mais leve, e fez dele não só um meio para que eu chegasse à tese, mas um fim. Obrigada.

Ao Otto por sua imensa capacidade de descrever situações complicadas com poucas palavras. Por sua presença marcante e estimulante. Pelos desafios lançados. Pelo exemplo dado.

Ao professor Leão pelo peso que seus ensinamentos tiveram na minha vida acadêmica.

Ao LuisH e ao Resende pela paciência e ajuda com os procedimentos burocráticos.

Aos professores Marcelo Rubinstein, Clicia e Satoru pela presença na banca examinadora.

Aos funcionários do Programa de Engenharia Elétrica da COPPE/UFRJ, Roberto, Maurício, Daniele e Solange pela presteza no atendimento na secretaria do Programa.

Ao José Antônio Pinto, que me permitiu chegar até o fim desta empreitada.

Ao CEFET-MG.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

UM PROTOCOLO DE TRANSPORTE COLABORATIVO PARA REDES DE SENSORES SEM FIO

Eugênia Cristina Müller Giancoli Jabour

Março/2009

Orientador: Aloysio de Castro Pinto Pedroza

Programa: Engenharia Elétrica

Este trabalho apresenta o CTCP, um novo protocolo de transporte para redes de sensores. Ele provê confiabilidade fim-a-fim e se adapta aos diferentes tipos de aplicação através de um mecanismo de variação da confiabilidade. Esta flexibilidade é obtida através de reconhecimentos salto a salto e de um algoritmo distribuído de confiabilidade dinâmica que se baseia na distribuição controlada de uma mesma mensagem sobre a rede. Observou-se que a recuperação de erros salto a salto e a duplicação da responsabilidade de armazenamento aumenta o percentual de mensagens corretamente transferidas. O controle e detecção de congestionamento proposto diferencia as perdas relativas a erro de transmissão daquelas relativas ao esgotamento de *buffers*. É chamado colaborativo já que todos os nós podem atuar sobre o controle de congestionamento e em função da responsabilidade distribuída de armazenamento. É escalável e independe das camadas de rede subjacentes. O aumento do consumo de energia, referente à introdução do CTCP na pilha de protocolos do sensor, foi calculado e discutido para os dois níveis de confiabilidade.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

COLLABORATIVE TRANSPORT CONTROL PROTOCOL FOR WIRELESS SENSOR NETWORKS

Eugênia Cristina Müller Giancoli Jabour

March/2009

Advisor: Aloysio de Castro Pinto Pedroza

Department: Electrical Engineering

This work presents the Collaborative Transport Control Protocol (CTCP), a new transport protocol for sensor networks. It aims at providing end-to-end reliability and adapts itself to different applications through a two level mechanism of reliability variation. CTCP achieves these properties using hop-by-hop acknowledgments and a storage control algorithm that operates at each node along a flow. It was observed that distributed fault recovery increases the average delivery rate and that duplication of storage responsibility minimizes the message loss. Its congestion control differentiates communication losses from buffer overflow. CTCP is called collaborative because all nodes detect and act on congestion control and also because it includes distributed storage responsibility. It is scalable and independent of the underlying network layer. The protocol energy consumption overhead was calculated and discussed for two reliability levels.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

As redes de sensores sem fio (RSSF) constituem um novo domínio da computação distribuída e têm sido alvo de grande interesse de pesquisa nos últimos anos. Fornecem uma solução de sensoriamento amplamente distribuída e econômica para ambientes onde as redes tradicionais não conseguiriam atuar. Uma RSSF é um tipo de rede *ad-hoc* formada por nós bastante limitados em termos de processamento, capacidade de comunicação e disponibilidade de energia. Em contrapartida, o baixo custo destes nós permite a construção de redes compostas por centenas ou milhares deles [2]. No entanto, a principal peculiaridade destas redes está nos seus objetivos [3]. Seus nós, dotados de módulos de sensoriamento, têm como finalidade realizar o monitoramento distribuído de uma região para um observador externo que é ligado à rede através de um nó sorvedouro(ou estação base). Desta forma, o objetivo da rede não é atender às necessidades dos nós que a compõem, mas sim às requisições deste observador externo.

Nestas redes, que podem ou não estar ligadas à Internet, os próprios nós sem fio são responsáveis pelo encaminhamento dos pacotes. Os nós devem, portanto, cooperar a fim de permitir a comunicação de múltiplos saltos entre o sorvedouro e os nós afastados, que não podem se comunicar diretamente. Por isso, elas apresentam alguns problemas peculiares. Entre estes, podem-se destacar a sensibilidade à interferência externa, a formação de áreas de sombra devido à existência de obstáculos e a variação da qualidade do sinal ao longo do tempo devido ao desvanecimento. Nas redes sem fio de múltiplos saltos [4], um planejamento abrangente capaz de reduzir significativamente estes problemas é complexo e muitas vezes inviável. Este tipo de

rede ainda apresenta um problema muito particular que é o consumo de energia dos nós, pois alguns cenários de aplicação destas redes prevêem a utilização de nós com um suprimento finito de energia. Nestes cenários, a conservação da energia dos nós é fundamental [5, 6, 7, 8, 9].

Suas principais aplicações são o monitoramento militar, ambiental, industrial ou de infra-estruturas domésticas.

1.1 Motivação

Ainda existe um caminho a ser trilhado antes de uma disseminação comercial destas redes. Alguns problemas precisam ser superados como, por exemplo, conservação de energia, controle de congestionamento, confiabilidade na disseminação dos dados, segurança e auto gerenciamento. Estes problemas, freqüentemente, envolvem uma ou mais camadas da arquitetura de redes e podem ser estudados separadamente em cada uma das camadas, ou colaborativamente através de todas elas.

Na camada física, por exemplo, o *hardware* das redes de sensores ainda é escolhido visando maximizar a vida útil da rede para uma aplicação específica. Contudo, observa-se que o número de aplicações para redes de sensores está crescendo e existe a necessidade de dispositivos mais poderosos e genéricos que poderão ser reprogramados durante sua vida útil. Como grande parte dos cenários de aplicação das redes de sensores sem fio supõe seu estabelecimento em regiões onde o acesso posterior aos nós sensores é difícil ou mesmo impossível, a conservação de energia e a reprogramação dos nós [10] são aspectos fundamentais no desenvolvimento de protocolos.

As redes de sensores podem ser utilizadas para implementar aplicações distintas. Entre estas aplicações, destacam-se as aplicações militares, as aplicações ligadas ao acompanhamento médico e as aplicações de vigilância. Embora todas estas aplicações sejam voltadas para a monitoramento de algum fenômeno ou evento, diferentes aplicações podem apresentar requisitos distintos de banda passante, atraso e processamento. Estas particularidades resultam em diferentes exigências a serem impostas aos protocolos das camadas mais baixas e, sobretudo, em diferentes padrões de consumo de energia [11].

As RSSF, tipicamente, operam sob pequena carga e subitamente se tornam ativas em resposta a um evento detectado. Dependendo da aplicação, isto pode resultar na geração de um grande e repentino fluxo de dados que deve ser entregue para um pequeno número de sorvedouros sem uma significativa degradação do desempenho da aplicação [12]. Constatou-se que o congestionamento costuma acontecer ao redor destes sorvedouros.

Pode-se entender uma situação de congestionamento quando as taxas de entrada e saída de dados, em um determinado ponto da rede, tornam-se desproporcionais de forma que o número de pacotes de entrada seja maior que o número de pacotes de saída, em um determinado nó da rede. Tal fato pode provocar um enfileiramento de pacotes no *buffer* de entrada do nó que normalmente reage a esta situação descartando os pacotes. Como conseqüência, poderão existir perdas definitivas de dados e aumento no tempo de resposta, degradando de forma geral o desempenho da rede [13].

Intuitivamente, uma grande quantidade de nós sensores instalados em um ambiente implicaria em uma maior precisão no sensoriamento do ambiente e uma maior quantidade de energia disponível na rede. Contudo, se não for gerenciada adequadamente, esta rede de alta densidade pode sofrer com o aumento do número de colisões de pacotes e potencialmente, com o surgimento de um congestionamento, que resulta em um aumento da latência na entrega dos pacotes e em um maior consumo de energia. Assim, alguns pesquisadores, recentemente, voltaram suas atenções para a camada de transporte, que é extremamente importante na confiabilidade dos dados disseminados e na conservação de energia.

A maioria dos protocolos existentes para a camada de transporte das redes de sensores foi adaptada para funcionar com determinados tipos de aplicações, ou assume que os nós trabalham com determinadas camadas de rede ou enlace. O protocolo ESRT (*Event-to-sink Reliable Transport*) [14], por exemplo, opera em redes de sensores que estão organizadas de forma plana [15] e sua metodologia de trabalho não garante a entrega fim-a-fim de dados. Seu objetivo principal é entregar um número representativo de pacotes de um determinado evento. Este comportamento pode não atender a uma classe de aplicações que requer confiabilidade. No projeto do protocolo proposto nesta tese, partiu-se do pressuposto que a entrega de cada

uma das mensagens é imprescindível e que o protocolo proposto deve suportar uma classe de aplicações que requer entrega de dados confiável nas redes de sensores sem fio.

O protocolo RMST (*Reliable Multi-Segment Transport*) [16] foi projetado para trabalhar em conjunto com o protocolo *directed diffusion* [17], ou seja, existe uma dependência da camada de rede. Neste trabalho, buscou-se projetar um protocolo independente das camadas subjacentes. O CODA (*Congestion Detection and Avoidance*) [12], por sua vez, não possui como objetivo a entrega confiável, mas sim a detecção e controle de congestionamento. Constatou-se, durante o desenvolvimento do protocolo proposto nesta tese, que a inclusão de um mecanismo de detecção e controle de congestionamento auxiliaria o mecanismo de entrega confiável e ainda diminuiria o consumo de energia.

Como resultado, estes protocolos não podem ser aplicados em qualquer tipo de rede de sensores. Portanto, é necessário projetar um protocolo da camada de transporte que possa suportar aplicações múltiplas na mesma rede, provendo controle do nível de confiabilidade, controle de congestionamento, redução de perdas e suporte a desconexões freqüentes.

1.2 Objetivos

O protocolo proposto por este trabalho [18, 19], Protocolo de Transporte Colaborativo *(Collaborative Transport Control Protocol - CTCP)*, oferece um transporte confiável, escalável e genérico às RSSF onde cada nó pode ser a origem de vários fluxos com diferentes características. Seu projeto se desenvolve principalmente em torno de dois núcleos: entrega confiável e controle de congestionamento.

O primeiro objetivo deste trabalho é prover entrega confiável ao sorvedouro (estação base) das mensagens geradas pela rede. Para isso, foi desenvolvido um **Algoritmo Distribuído de Confiabilidade Dinâmica** (ADCD) que se baseia na distribuição controlada do armazenamento de uma mesma mensagem pela rede. Este algoritmo trabalha com um compromisso entre confiabilidade e consumo de energia. Desta maneira, o protocolo é capaz de se adaptar a aplicações com alta exigência de entrega e àquelas que não exigem tanto. Quanto mais alta a taxa de

entrega requerida pela aplicação maior é o seu consumo de energia. A capacidade de adaptação do protocolo é estendida por sua independência das camadas subjacentes. Tal flexibilidade confere ao protocolo uma maior possibilidade de ser utilizado. Entretanto, o ponto forte deste algoritmo concentra-se na sua grande capacidade de recuperação, e entrega, mesmo na presença de falhas simultâneas de nós. Esta funcionalidade é alcançada através da duplicação de reconhecimentos positivos (ACKs).

O segundo objetivo deste protocolo é detectar e controlar o congestionamento geral da rede. Para tanto, criou-se um mecanismo capaz de interromper o tráfego da rede e liberá-lo rapidamente ao detectar iminência de congestionamento. Tal mecanismo mostrou-se capaz de diferenciar uma perda relativa a congestionamento de uma relativa a erro de transmissão. A liberação do *buffer* dos nós gerada pelo ADCD também contribui para manter a rede com baixo consumo e sem congestionamentos.

1.3 Estrutura do Documento

A tese está organizada como se segue.

O Capítulo 2 apresenta uma visão geral das redes de sensores, focando, principalmente, na camada de transporte. São apresentados conceitos relacionados à arquitetura e ao funcionamento destas redes. Aborda os requisitos da camada de transporte e discute os trabalhos relacionados ao protocolo proposto nesta tese.

Um protocolo de transporte colaborativo para RSSF é proposto no Capítulo 3. O protocolo proposto é dividido em duas partes: o algoritmo distribuído de confiabilidade e o algoritmo para detecção e controle de congestionamento. São apresentados dois níveis distintos de confiabilidade, que podem ser alternados dinamicamente. O primeiro oferece garantia de entrega de cada uma das mensagens geradas. O segundo oferece uma confiabilidade limitada a um menor custo energético. Para um funcionamento ideal destes dois algoritmos usou-se o algoritmo de karn [20] como base para os dois intervalos de controle do temporizador. Deste modo, gerou-se uma calibragem do algoritmo de karn para as redes de sensores sem fio. Em seguida é proposto o algoritmo de detecção e controle de congestionamento que trabalha em conjunto com os algoritmos de confiabilidade visando minimizar perdas de mensagens. Ao final do capítulo, encontra-se a especificação formal destes algoritmos.

O Capítulo 4 foi desenvolvido em paralelo ao projeto do protocolo descrito no Capítulo 3. As idéias foram modeladas, implementadas e testadas, visando verificar sua viabilidade. Portanto, o capítulo começa com uma comparação entre dois paradigmas de reconhecimento: fim-a-fim e salto a salto. Em seguida, avalia-se probabilisticamente os dois níveis de confiabilidade propostos em relação às métricas "sucesso do protocolo" e "entrega de mensagens". Além dos modelos matemáticos, optou-se pelo desenvolvimento, em java [21], de um simulador probabilístico. Sua descrição e utilização são detalhadas, também, neste capítulo. O código fonte do simulador encontra-se no Apêndice A.

O Capítulo 5 começa pela apresentação do sistema operacional TinyOS [22, 23] e do simulador TOSSIM [24]. A seguir, faz-se um detalhamento da implementação do CTCP. O código fonte do protocolo encontra-se no Apêndice B.

As simulações são descritas no Capítulo 6. Após a concepção do protocolo, foi feita uma avaliação estatística baseada na análise da confiabilidade de sistemas distribuídos. Esta análise, contida no Capítulo 4, nos forneceu parâmetros de configuração e direcionamentos de desenvolvimento que foram usados durante a etapa de simulação. Durante esta etapa, avaliou-se o mecanismo de confiabilidade através de três métricas: entrega de pacotes, latência e consumo de energia. O algoritmo de controle e detecção de congestionamento foi analisado levando-se em conta o número de perdas antes e depois da inserção deste mecanismo.

Por fim, o Capítulo 7 conclui esta tese e apresenta possíveis direções futuras de pesquisa.

Capítulo 2

As Redes de Sensores Sem Fio

Este capítulo apresenta uma visão geral das redes de sensores, focando, principalmente, na camada de transporte. São apresentados conceitos relacionados à arquitetura e ao funcionamento destas redes. Aborda os requisitos da camada de transporte e discute os trabalhos relacionados ao protocolo proposto nesta tese.

2.1 Conceitos Principais

As Redes de Sensores Sem Fio (RSSF) se apresentam de várias formas [25, 26, 27, 28, 29, 30]. Cobrem diferentes áreas geográficas, podem ser esparsas ou densamente distribuídas, usam dispositivos com grandes restrições energéticas e implementam aplicações de diferentes classes. Portanto, possuem características peculiares bem distintas das redes tradicionais, principalmente no que diz respeito a quantidade de nós presentes, poder de processamento, memória disponível e largura de banda de comunicação. Estas variáveis tornam praticamente impossível a reutilização de vários algoritmos desenvolvidos para os sistemas computacionais tradicionais, uma vez que o projeto de qualquer solução para redes de sensores deve considerar o consumo de energia.

Sumarizando, segundo Chiang [31], os requisitos gerais para as redes de sensores são:

- **Baixo Consumo de Energia** - O nós sensores, normalmente, são alimentados a bateria. A substituição manual das baterias geralmente não é possível, o que torna os nós dependentes da vida útil delas. Como resultado, a mini-

mização do consumo de energia torna-se crítica quando se pretende alcançar um sistema robusto;

- **Escalabilidade** - A ordem de grandeza do número de nós de uma rede de sensores sem fio pode variar das dezenas aos milhares. Os novos protocolos devem ser capazes não somente de lidar com este número de nós, mas, também, de utilizá-los em todo o seu potencial. Devem, ainda, considerar a densidade com que os sensores estão espalhados na região a ser sensoriada. Esta densidade pode variar muito e os novos esquemas de transmissão devem ser capazes de lidar com esta variação e utilizá-la a seu favor;

- **Habilidade de Auto-Organização** - As redes de sensores podem ser grandes em tamanho e o seu trabalho em ambientes hostis pode aumentar o número de falhas individuais dos nós. São necessários mecanismos para reorganizar a rede depois destas falhas. Assim, a capacidade de auto-organizar-se torna-se essencial.

2.2 Arquitetura da Rede

Quanto à sua estrutura, uma RSSF é organizada como um sistema com três componentes principais: (1) infra-estrutura; (2) pilha de protocolos e (3) aplicação [32]. A infra-estrutura refere-se aos sensores físicos (suas características e capacidades), ao número de sensores e à sua estratégia de instalação (como e onde são instalados). A pilha de protocolos refere-se ao software que implementa as diversas camadas de protocolos existentes em cada nó da rede. O componente de aplicação representa os interesses ou consultas do usuário, que são traduzidos em tarefas de sensoriamento a serem executadas pela rede. Vários procedimentos de otimização envolvendo os três componentes organizacionais citados podem ser empregados da RSSF com o objetivo de aumentar seu desempenho.

2.2.1 Infra-estrutura

A infra-estrutura de uma RSSF consiste nos nós da rede e no seu estado de instalação no ambiente. Em geral, os nós sensores são espalhados por uma região de difícil

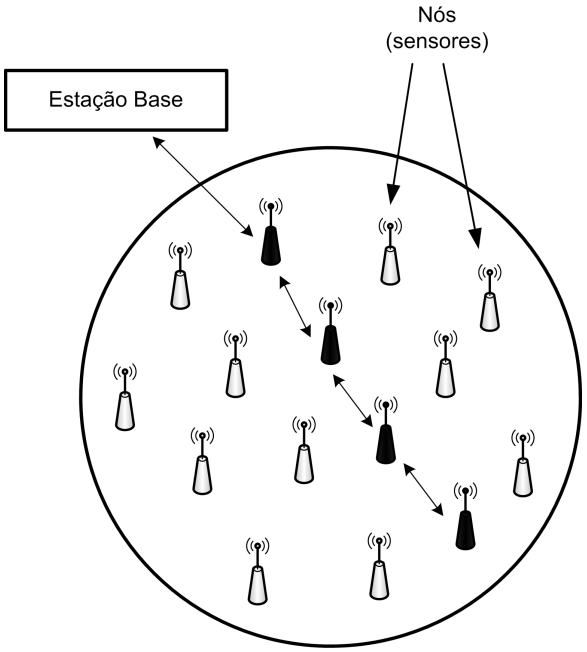

Figura 2.1: Rede de Sensores Sem Fio.

acesso, formando uma rede de sensores sem fio, como na Figura 2.1. Cada um dos sensores pode possuir vários dispositivos sensoriais que são usados para coletar dados. Os dados coletados podem ser processados ou não, de acordo com a aplicação, e devem ser escoados para a estação base. Uma rede pode possuir um ou mais pontos de escoamento. Esses nós fazem a interface entre a aplicação e a rede, servindo de ponto de entrada para a submissão dos interesses da aplicação e de concentrador dos dados enviados pelos nós sensores. São nós com maior poder computacional e sem restrições de energia. Nós sensores possuem capacidades de processamento e armazenamento limitadas. Sua função é coletar, eventualmente agregar e transmitir seus próprios dados e os dos seus vizinhos para a estação base.

Nós Sensores

Os nós sensores são constituídos de quatro componentes principais: uma unidade de sensoriamento, uma unidade de processamento, um transceptor (*transceiver*) e uma fonte de energia. Todos estes componentes devem ser interligados, como mostra a Figura 2.2, para permitir o processamento local e a transmissão das informações.

Dependendo da aplicação, é possível que os sensores sejam dotados de unidades específicas adicionais, como, por exemplo, unidades de localização ou mobili-

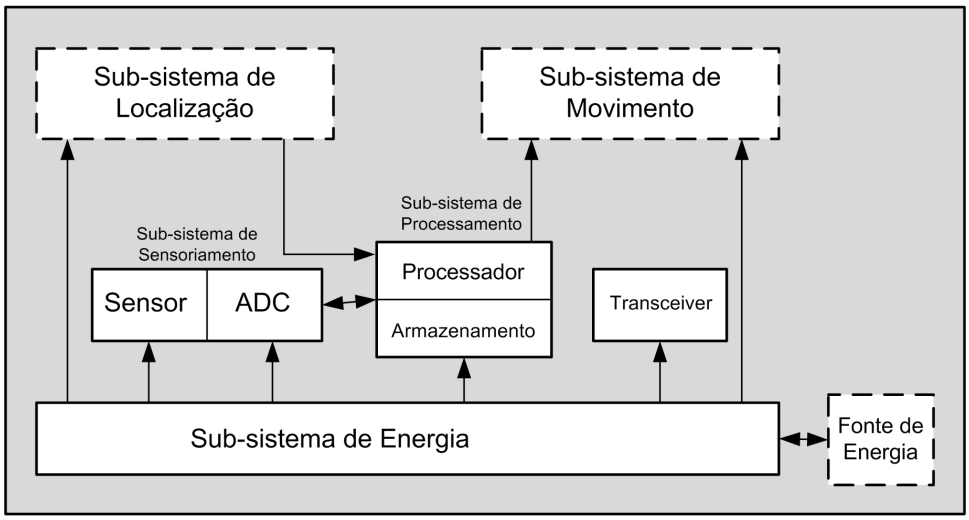

Figura 2.2: Componentes do *hardware* de um nó sensor [1].

zadoras [15]. O subsistema de sensoriamento é geralmente composto pelos dispositivos de sensoriamento e pelo conversor de sinal analógico para digital (ADC). Uma RSSF pode conter muitos tipos diferentes de dispositivos de sensoriamento, como sísmicos, magnéticos, térmicos, visuais, infravermelhos, acústicos e radares. Os sinais analógicos produzidos pelos sensores a partir do fenômeno monitorado são convertidos em sinais digitais pelo ADC e são passados para o processador. O subsistema de processamento, que em geral está associado à unidade de armazenamento local, é responsável: (1) pela execução dos protocolos de comunicação e dos algoritmos de processamento de dados; (2) pelo controle dos sensores; e (3) pela gerência dos procedimentos necessários para que os nós atuem de forma colaborativa.

O subsistema de comunicação normalmente é baseado em rádio freqüência e é composto por um transceptor (*transceiver*), uma antena e um conjunto de componentes discretos para configurar as características da camada física, como sensibilidade e intensidade do sinal. Em geral, o transceptor pode operar em três modos: recepção, transmissão e desligado [1].

A fonte de energia dos sensores consiste de uma bateria e um conversor DC-DC. A bateria é um dispositivo complexo, cuja operação depende de diversas características como, por exemplo, suas dimensões, o material do qual é feita e a taxa de descarga. O conversor DC-DC é responsável por fornecer uma tensão constante para o sensor e o seu fator de eficiência determina o tempo de vida da bateria [33].

Além dos componentes de hardware descritos anteriormente, o projeto de um nó

sensor pode incluir um sistema operacional. Quando presente, o sistema operacional deve gerenciar a operação do nó sensor da forma mais eficiente possível. Um exemplo de sistema operacional desenvolvido especialmente para sensores, e utilizado em grande parte do hardware hoje existente, é o TinyOS [22]. Este sistema operacional está descrito no Capítulo 5.

Instalação no ambiente

O estado de instalação da rede diz respeito à localização dos sensores no espaço físico, à densidade da rede e a possíveis deslocamentos no caso de sensores móveis. Centenas a milhares de nós sensores são instalados na área a ser monitorada, distanciados no máximo de algumas dezenas de metros uns dos outros. Gerenciar um elevado número de nós, em uma rede com alta densidade, requer uma criteriosa tarefa de manutenção de topologia por parte dos protocolos da rede. A alta probabilidade de falhas ou o possível deslocamento dos sensores dificultam essa tarefa. Nos trabalhos de Jabour *et al.* [34, 35, 36, 37] foram examinadas algumas questões relacionadas à mobilidade dos nós e às mudanças na topologia.

2.2.2 Pilha de Protocolos

As redes de sensores sem fio, como outras redes, são baseadas no modelo de referência ISO OSI [38]. A estação base e os nós utilizam a pilha de protocolos ilustrada na Figura 2.3. Esta pilha de protocolos é composta pelas camadas de aplicação, transporte, rede, enlace de dados e física. Contudo, é importante ressaltar que, dependendo da aplicação, pode-se optar pela utilização parcial da pilha. Por exemplo, em uma rede de sensores distribuída por uma pequena área geográfica, qualquer nó pode alcançar, em um único salto, a estação base. Desta maneira, não é necessária a utilização de algoritmos de roteamento, o que dispensa a utilização da camada de rede. Cada uma destas camadas está descrita a seguir.

- **Camada Física** - A camada física é responsável por transmitir bits através da utilização de técnicas de modulação, sobre as bandas de freqüência alocadas. Nas redes de sensores sem fio, geralmente, são utilizados esquemas de modulação simples tais como *Binary Phase Shifting Keying (BPSK)* ou *QPSK*,

Figura 2.3: Pilha de protocolos utilizada pelos nós sensores.

que são suficientes para prover baixas taxas de dados. O esquema *Direct Sequence Spread Spectrum (DSSS)* também é usado. As bandas de freqüências mais utilizadas são aquelas não licenciadas, na faixa de 900 ou 2400 MHz, ou as ondas de infravermelho para a comunicação em linha de visada [31].

- **Camada de Enlace de Dados** - A camada de enlace de dados é a camada responsável pela transmissão confiável de dados, salto a salto. Esta camada fornece o protocolo necessário para o acesso e compartilhamento do canal de comunicação. Devido ao consumo de energia e a requisitos de auto-organização, os protocolos convencionais de acesso ao meio são evitados, e conseqüentemente, novos protocolos foram propostos [39, 40].

- **Camada de Rede** - Utiliza técnicas de roteamento eficientes, que são essenciais para preservar a energia [41, 42, 43, 44, 45]. O ambiente operacional descontrolado, com falhas randômicas dos nós, aumenta a complexidade do roteamento.

- **Camada de Transporte** - A camada de transporte é responsável por proporcionar a entrega de dados confiável fim-a-fim, ou seja, entre a origem e o sorvedouro. Esta camada está detalhada neste capítulo.

- **Camada de Aplicação** - A camada de aplicação provê vários serviços para as aplicações das redes de sensores: serviço de localização, agregação de dados, desligamento de sensores, sincronização do tempo e configuração da rede.

2.2.3 Aplicações

Nós sensores podem ser usados para o sensoriamento contínuo de dados, para a detecção de eventos e para o controle local de atuadores, além de outras possíveis tarefas. Desse modo, o uso de RSSFs habilita uma ampla gama de aplicações potenciais, pertencentes a diversos ramos do conhecimento humano. A aplicação é o componente de uma arquitetura de RSSF responsável por emitir um conjunto de consultas ou interesses, que descrevem as características dos fenômenos físicos que o usuário deseja analisar. Os interesses da aplicação devem indicar os tipos de dados desejados; a freqüência com que esses dados devem ser coletados; a necessidade ou não dos dados sofrerem algum tipo de agregação; os requisitos de QoS, como valores de atraso máximo ou acurácia mínima desejados; os limiares a partir dos quais os dados devem ser transmitidos; ou ainda, eventos que podem disparar algum comportamento particular da rede, como a ativação de sensores específicos ou a alteração na taxa de sensoriamento [1].

As aplicações potenciais para RSSFs podem ser categorizadas nas seguintes áreas: militar, ambiental, de saúde, doméstica e comercial. É possível expandir essa classificação com categorias adicionais, como a exploração espacial, o processamento químico e o socorro a desastres [46].

2.2.4 Modelos de Comunicação e Entrega de Dados

Conceitualmente, a comunicação em uma rede de sensores pode ser classificada em duas categorias: aplicação e infra-estrutura. O protocolo de rede deve suportar estes dois tipos de comunicação [15]. A comunicação da aplicação é responsável pela transferência de interesses da aplicação com o objetivo de informar ao observador sobre um dado fenômeno. Idealmente, o interesse da aplicação deve ser especificado em termos do fenômeno em estudo, evitando que o usuário tenha que conhecer a infra-estrutura e os protocolos de comunicação subjacentes. A comunicação de infra-estrutura é responsável pela troca de mensagens entre os nós. Ela é necessária para configurar, manter e otimizar a operação da rede. Neste tipo de comunicação estão incluídas, por exemplo, as trocas de mensagens para descobrir os caminhos dos sensores para a estação base, desconsiderando a mobilidade ou falha do sensor. Segundo Pereira *et al.* [15], como a infra-estrutura de comunicação representa o

overhead do protocolo, é importante minimizar esta comunicação, assegurando que a rede possa suportar a comunicação da aplicação de forma eficiente.

Quanto ao modelo de entrega de dados, o qual é ditado pela aplicação, as RSSFs podem ser classificadas em contínuas, dirigidas a eventos, iniciadas pelo observador ou híbridas. No modelo contínuo, ou proativo, sensores comunicam seus dados continuamente a uma taxa pré-definida. No modelo dirigido a eventos, ou reativo, os sensores reportam informações somente se um evento de interesse ocorrer. Nesse caso, a aplicação está interessada apenas na ocorrência de um fenômeno específico. No modelo iniciado pelo observador, os sensores reportam seus resultados em resposta a um pedido explícito (síncrono) da aplicação. A aplicação está interessada em obter uma visão instantânea do fenômeno monitorado. Finalmente, as três abordagens podem coexistir na mesma rede, gerando um modelo híbrido de entrega.

Uma vez definido o modelo de entrega de dados a ser usado na rede para atender a uma determinada aplicação, protocolos que encaminhem de forma eficiente os dados desde sua origem (nó fonte) até seu destino (nó sorvedouro ou estação base) devem ser empregados.

2.3 Desafios da Camada de Transporte

Este trabalho propõe soluções para criar uma entrega de dados confiável nas RSSF e por isso foca seus esforços na camada de transporte cujo principal objetivo é oferecer um serviço confiável, eficiente e econômico a seus usuários que, em geral, são processos presentes na camada de aplicação. A independência da rede física ou das camadas subjacentes é um requisito importante, à medida que possibilita uma maior utilização do protocolo em diferentes cenários.

Uma questão importante a ser considerada é o compromisso entre a implementação da confiabilidade salto a salto na camada de enlace ou na de transporte, que, se concentra em prover confiabilidade fim-a-fim. De acordo com Wang *et al.* [47], mesmo que o protocolo MAC (da camada de enlace) possa recuperar pacotes perdidos através do mecanismo correção de erros de bits, ele, normalmente, não possui maneiras de tratar os pacotes descartados por *buffer* cheio. Logo, o protocolo de transporte das RSSF deve possuir mecanismos de reconhecimento (ACK e/ou *Se-*

lective ACK [48] para recuperar os pacotes perdidos).

A fim de detectar e recuperar pacotes perdidos nas RSSF, um dos aspectos a se considerar, com cuidado, é como detectar o congestionamento e como superá-lo. As soluções fim-a-fim são bastante simples e robustas, mas injetam muitos pacotes na rede. Em contrapartida, as soluções salto a salto injetam menos pacotes e podem rapidamente enfraquecer um congestionamento. Contudo, nesta última abordagem, o comportamento de cada nó, entre a origem e o destino, precisa ser alterado. Como uma menor quantidade de pacotes na rede pode resultar em economia de energia, existe um compromisso entre os mecanismos salto a salto e fim-a-fim, que deve ser cuidadosamente considerado ao se projetar algoritmos de controle de congestionamento para as RSSF [47].

As redes tradicionais garantem confiabilidade na camada de transporte através de mecanismos de recuperação de erros fim-a-fim [49]. Esta abordagem não é ideal para as redes de sensores pois, ao contrário das redes tradicionais onde os nós intermediários são apenas roteadores, nas redes de sensores eles possuem a camada de transporte. O fato de cada nó possuir a camada de transporte possibilita a distribuição da tarefa de recuperação de erros para cada um destes nós. Esta colaboração entre os nós torna-se factível porque todos os nós pertencem à mesma entidade administrativa e podem ser utilizados para atingir um mesmo objetivo.

De acordo com os autores de [47, 50, 51], os requisitos básicos para uma camada de transporte genérica para as redes de sensores são:

- **Heterogeneidade** - Os nós sensores podem possuir múltiplos sensores (luz, temperatura, presença, etc) com características diferentes de transmissão. Os pacotes gerados por cada sensor para uma aplicação constitui seu fluxo de dados, que pode ser contínuo ou baseado em eventos. Nas aplicações de fluxos contínuos, os nós transmitem pacotes periodicamente para a estação base. Nas aplicações baseadas em evento, os nós transmitem dados somente quando um determinado evento ocorrer. Os dois tipos de fluxo podem existir na mesma rede e o protocolo da camada de transporte deve suportar múltiplas aplicações heterogêneas dentro da mesma rede;

- **Confiabilidade fim-a-fim** - As aplicações têm requisitos diferentes de confiabilidade. Por exemplo, em ambiente militar, os dados transmitidos pelos

sensores devem chegar sempre à estação base. Na reprogramação de grupo de sensores, os dados também precisam chegar até todos os nós. Entretanto, no monitoramento de temperatura, alguns pacotes podem ser perdidos sem causar grandes danos. O protocolo de transporte deve explorar estas diferenças, objetivando economizar energia;

- **Controle de congestionamento** - Todos os nós da rede geram pacotes que convergem para a estação base.Oos nós localizados ao redor da estação base encaminham um maior número de pacotes o que, conseqüentemente, aumenta a possibilidade de congestionamento perto da estação base. Altas taxas de dados, rajadas de dados e colisões são as outras razões para os congestionamentos nas redes de sensores;

- **Controle de Fluxo** - É preciso usar um mecanismo de controle para evitar que um transmissor rápido sobrecarregue um receptor lento. Protocolos salto a salto estão aptos a controlar cada um dos saltos ao longo de uma rota de rede, o que, de acordo com Heimlicher *et al.* [50], é uma clara vantagem sobre os protocolos fim-a-fim;

- **Simplificação da conexão inicial** - Os protocolos de transporte para RSSF devem simplificar o processo de abertura de conexão ou usar protocolos sem conexão, a fim de começar a transmissão de dados o mais rápido possível e sobretudo, garantir alta vazão e atraso mínimo. Algumas aplicações em RSSF são reativas a certos eventos específicos e esperam que estes eventos ocorram para começar a transmitir para a estação base. Estas aplicações possuem poucos pacotes para cada ocorrência do evento e, assim, um processo de abertura de conexão rápido seria mais efetivo e eficiente.

A perda de pacotes está sempre presente nas RSSFs devido à má qualidade dos canais sem fio, às falhas dos sensores e ao congestionamento de pacotes ao redor da estação base. As RSSFs devem garantir algum grau de confiabilidade aos pacotes ou às mensagens da camada de aplicação para conseguir obter informações íntegras da rede. Algumas aplicações críticas necessitam de transmissão confiável para cada pacote e, conseqüentemente, uma camada que aumente a garantia de entrega de

pacotes passa a ser necessária. De qualquer maneira, o primeiro passo consiste em detectar a perda de pacotes a fim de poder recuperá-los.

Métodos utilizados em troca de pacotes nas redes tradicionais podem ser aproveitados nas RSSF. Por exemplo, os mecanismos de reconhecimento (ACK ou NACK) podem ser usados para recuperar pacotes perdidos tanto nas abordagens fim-a-fim quanto nas abordagens salto a salto. Intuitivamente, se existem um menor número de pacotes na rede e poucas retransmissões, a economia de energia aumenta. Deste modo, a fim de reduzir o número de pacotes na rede utilizam-se controles de congestionamento além de uma recuperação eficiente das perdas que pode resultar em poucas retransmissões. Assim, o controle de congestionamento eficiente e a garantia de confiabilidade resultam em economia de energia nas redes de sensores.

Este trabalho propõe o CTCP. Trata-se de um protocolo de transporte colaborativo baseado em mecanismos conhecidos de reconhecimentos (ACK) e temporizadores [48]. Os dois níveis de confiabilidade propostos por esta tese garantem flexibilidade ao CTCP e possibilitam a sua adaptação a diferentes tipos de aplicação. Este mecanismo objetiva suportar interrupções de conexão sem perda de dados. Mesmo quando um nó recebe os dados a serem transmitidos e falha antes de reencaminhá-los, o protocolo está apto a recuperar esta perda. O CTCP usa reconhecimentos salto a salto, considerados eficientes por Stann *et al.* [16], com liberação imediata do *buffer*, o que aumenta a capacidade de reencaminhar pacotes e previne o congestionamento. Além disso, o protocolo prevê um mecanismo de controle de congestionamento apto a evitar perdas relativas a *buffer* cheio. O CTCP foi projetado para trabalhar com quaisquer camadas subjacentes.

O serviço de transporte é implementado por um protocolo usado entre duas entidades de transporte. Em alguns aspectos, os protocolos de transporte [48, 52] possuem funções similares aos protocolos de enlace de dados, pois ambos têm de lidar com o controle de erros, com a definição de seqüências e com o controle de fluxo, entre outros itens [53]. Entretanto, existem diferenças significativas entre os dois. Essas diferenças ocorrem devido às peculiaridades dos ambientes nos quais os dois protocolos operam. Na camada de enlace de dados os dois roteadores se comunicam diretamente através do canal físico, enquanto na camada de transporte esse canal físico é substituído pela sub-rede inteira. Na camada de enlace de dados,

o roteador não precisa especificar com que roteador deseja se comunicar, pois cada linha de saída especifica de modo exclusivo um determinado roteador. Na camada de transporte, é necessário o endereçamento explícito de destinos.

2.4 Protocolos de Transporte Existentes

O primeiro protocolo analisado foi o TCP [48, 54]. Este protocolo faz parte da pilha de protocolos TCP/IP que foi teoricamente projetada para operar de forma independente das tecnologias das camadas inferiores. Assim, o perfil de protocolos TCP/IP deve operar em redes cabeadas confiáveis, redes sem fio, redes de satélite e redes ópticas. No entanto, os atuais mecanismos do TCP se baseiam em suposições típicas de redes cabeadas convencionais, tais como a existência de uma conectividade fim-a-fim entre fonte e destino durante todo o período correspondente à sessão de comunicação, atrasos de comunicação relativamente pequenos (na ordem de milissegundos), baixas taxas de erros, mecanismos de retransmissão efetivos para reparar erros e suporte a taxas de dados bidirecionais relativamente simétricas.

Desta forma, o TCP não se adequa às redes de sensores. Estas redes são caracterizadas por atrasos longos ou variáveis, perda freqüente de conexões, conectividade intermitente, altas taxas de erro e limitação de recursos. A pilha TCP/IP apresenta um baixo desempenho nestas redes.

O protocolo *Reliable Multi-Segment Transport* (RMST) [16] foi projetado para funcionar em conjunto com o *Direct Diffusion Protocol* [17], ou seja, existe uma dependência em relação à camada de rede. O RMST é um protocolo baseado em NACK e trabalha com ou sem *cache*. Quando o *cache* está habilitado, os nós intermediários armazenam os fragmentos de dados, o que pode causar esgotamentos do *buffer*. O RMST não garante a confiabilidade quando um nó falha depois de receber e antes de transmitir os fragmentos de dados. Além disso, o RMST não trata o congestionamento de dados na rede de sensores.

O protocolo *Event-to-Sink Reliable Transport* (ESRT) [14] foi projetado para redes centradas em aplicações. É pressuposto que a estação base está interessada em um determinado evento que pode ser detectado por vários nós ao mesmo tempo. No ESRT, é a estação base que faz o controle do congestionamento, solicitando aos

nós o aumento ou diminuição da taxa de transmissão. Não garante entrega fim-a-fim. Em redes reais, os nós só transmitem dados quando detectam um evento, o que dificulta o controle da taxa de transmissão pela estação base.

O protocolo *Pump Slowly, Fetch Quick* (PSFQ) [55] tem como objetivo reprogramar os nós de uma rede de sensores. Possibilita um *broadcast* de dados confiável da estação base para os nós. Contudo, possui alto consumo de energia, uma vez que a confiabilidade é alcançada através do aumento de retransmissões na rede.

O protocolo *Sensor Transmission Control Protocol* (STCP) [51] foi projetado para ser um protocolo genérico. Entende-se por genérico a capacidade de se adaptar a qualquer tipo de aplicação e trabalhar com qualquer camada de rede subjacente. Possui controle de congestionamento, uma vez que os nós ao redor da estação base podem estar sujeitos a esgotamento de *buffer*. Possui nível de confiabilidade adaptável visando economia de energia. Contudo, quase todos os seus controles baseiam-se fortemente na estação base, que possui amplos poderes computacionais e energéticos. Considera-se questionável que o nível de confiabilidade requerido pela aplicação seja controlado pelo nó que origina a informação, pois o conhecimento global da rede e da aplicação seria necessário para tal decisão. O sincronismo de tempo da rede de sensor é requerido para economizar energia em aplicações que possuam fluxos de dados contínuos. Contudo, não existem resultados que provem que o *overhead* causado pela sincronização justifique esta escolha.

Este trabalho utiliza o reconhecimento salto a salto, demonstrado eficiente por Stann *et al.* [16], mas prevê a liberação imediata do *cache* (*buffer*) do nó, aumentando a sua capacidade de reencaminhamento e evitando o congestionamento. O CTCP dispensa ainda a sincronização da rede, utilizada por Iyer [51] e é capaz de suportar falha das conexões sem perda de dados. Mesmo quando um nó recebe os dados e falha antes de reencaminhá-los, o protocolo é capaz de recuperar-se desta perda. Foi projetado para trabalhar com qualquer tipo de camada subjacente e possui um controle de congestionamento capaz de evitar perdas relativas a esgotamento de *buffer*.

2.5 Resumo

Neste capítulo fez-se uma revisão bibliográfica, focando principalmente os protocolos de transporte para redes de sensores sem fio. Constatou-se que os protocolos de transporte existentes envolvem uma parametrização especializada, sincronização de tempo entre os sensores ou uma profunda ligação com as camadas subjacentes. Existe uma demanda por protocolos especificamente projetados para:

- atender a uma classe de aplicações que requer entrega confiável de dados;
- possuir independência das camadas inferiores;
- recuperarem-se quando da falha de sensores;
- evitar perdas de segmentos por esgotamento de *buffer*;
- prevenir e reagir aos congestionamentos.

No próximo capítulo, o protocolo CTCP que incorpora todas as características citadas anteriormente, é descrito e especificado .

Capítulo 3

Descrição e Especificação do Protocolo CTCP

Neste capítulo, é apresentada a especificação do protocolo proposto nesta tese, o *Collaborative Transport Control Protocol* (CTCP). Seu principal objetivo é garantir a entrega fim-a-fim, de pacotes, nas redes de sensores sem fio. Para alcançar este objetivo, o protocolo foi composto por dois grandes núcleos. O primeiro propõe um algoritmo distribuído com dois níveis de confiabilidade, capaz de suportar a falha de um nó (nível 1) ou mesmo falhas simultâneas (nível 2). A estimativa do intervalo de controle do CTCP (temporizador) foi calibrada até que chegássemos aos resultados apresentados na Seção 3.3.2. O segundo núcleo do protocolo foca a prevenção e o controle de congestionamento na rede. Além destes dois algoritmos, são especificadas as demais funcionalidades do protocolo. Ao final temos a especificação formal do CTCP.

3.1 Premissas

Este trabalho pressupõe uma rede de sensores sem fio distribuída aleatoriamente em uma área de difícil acesso. Cada nó é equipado com um ou mais dispositivos de sensoriamento, um processador de baixo poder computacional e um transceptor(*transceiver*) sem fio de baixo alcance. A alimentação de cada nó é feita por bateria esgotável. Os nós são pré-configurados com um identificador único. Não existem requisitos para a estação base, uma vez que suas tarefas são equiparáveis

em complexidade às de um nó comum. Os nós sensores e a estação base se comunicam por enlaces sem fio bidirecionais. Não existe necessidade de sincronização de relógios entre os nós. A rede está pré-configurada com uma aplicação que requer entrega confiável de dados, mas pode ser reprogramada a qualquer momento. O fluxo de dados pode ser contínuo, orientado a evento ou iniciado pelo usuário. A camada de transporte proposta nesta tese independe das camadas adjacentes e pode trabalhar com qualquer protocolo de roteamento. Entretanto, o protocolo de roteamento, qualquer que seja ele, deve estar presente.

Na seqüência deste trabalho, são utilizados os termos mensagem, segmento e pacote como sinônimos.

3.2 Objetivos do CTCP

Os principais objetivos e características do CTCP são:

- garantir a entrega dos segmentos à camada de aplicação da estação base, mesmo na presença de falhas de nós e freqüentes desconexões;
- ser capaz de diferenciar a perda relativa a congestionamento da perda relativa a erro de transmissão, eliminando a primeira e solucionando a segunda através de reconhecimentos e retransmissões;
- controlar o congestionamento através da interrupção/liberação imediata do encaminhamento;
- ser independente das camadas subjacentes;
- oferecer dois perfis de confiabilidade visando economizar energia.

3.3 Operação do CTCP

3.3.1 Abertura e Fechamento da Conexão

Antes de iniciar a transmissão de dados, um pacote de abertura de conexão (ABR) é enviado, salto a salto, da origem para a estação base. Este pacote informa à estação base o identificador do fluxo de dados e o primeiro número de seqüência.

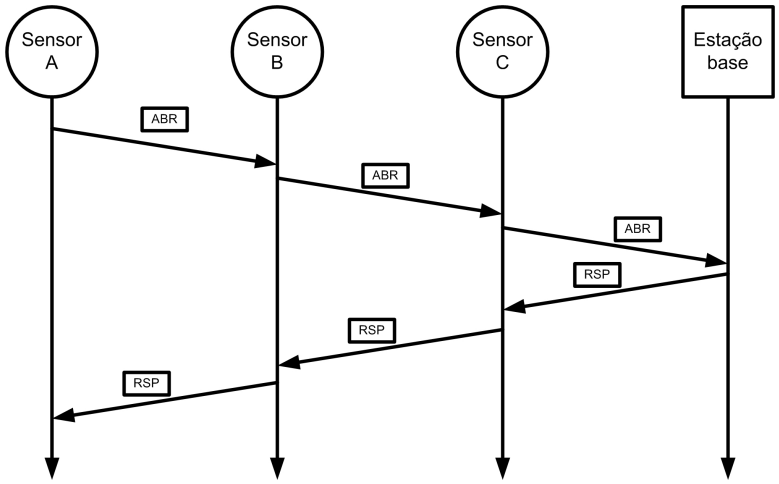

Figura 3.1: Abertura da conexão do protocolo CTCP.

Quando a estação base recebe este pacote, ela reserva os *buffers*, inicializa as variáveis necessárias e envia uma mensagem de resposta (RSP) à origem da conexão. No cabeçalho do RSP são especificados o nível de confiabilidade requerido pela aplicação à qual o fluxo pertence e o identificador da conexão (ID). Manter o identificador da conexão unívoco provê flexibilidade ao protocolo, uma vez que este dado é necessário para implementar a multiplexação de diferentes fluxos da rede. A multiplexação de fluxos será tratada em trabalhos futuros. A Figura 3.1 ilustra a fase de abertura da conexão.

Logo a seguir, o nó origem está apto a iniciar a transmissão de dados para a estação base. Para que este protocolo fosse o mais genérico possível, durante seu projeto, preocupou-se em estabelecer formatos de pacotes compatíveis com a maioria das redes subjacentes. Desta forma, foi preciso considerar que a comunicação sem fio, e principalmente as redes de sensores, possuem dificuldades de transferência de pacotes que sejam maiores do que o quadro da camada de enlace. Apesar de alguns protocolos, como o 802.11 [56], possuírem fragmentação e agrupamento, existem limites no tamanho dos pacotes que uma entidade consegue fragmentar e garantir a entrega [16]. Por isso, os sensores que podem utilizar o CTCP são pré-configurados com o MSS (*Maximum Segment Size*) permitido pelas camadas de rede subjacente. Tal variável não precisa ser negociada durante a abertura da conexão.

Esta troca inicial de mensagens é feita utilizando-se o nível 1 de confiabilidade até que o pacote RSP chegue ao nó origem com o novo nível de confiabilidade

determinado para aquele momento.

Quando a camada de aplicação do nó origem termina seu trabalho, envia um pacote de fechamento de conexão (CLO) à estação base. A estação base, então, libera o *buffer* e variáveis da conexão.

3.3.2 Algoritmo Distribuído de Confiabilidade Dinâmica

Cada aplicação possui requisitos diferentes de confiabilidade. Algumas, por exemplo, suportam perdas de dados, enquanto outras precisam garantir que cada um dos seus pacotes chega ao destino. Desta forma, para especificar a confiabilidade requerida, é preciso que se tenha conhecimento da aplicação e de seus objetivos. É preciso ressaltar que o nível de confiabilidade, configurado durante a fase de abertura de conexão, pode ser alterado a qualquer momento pelo usuário que interage com a estação base. A necessidade de alteração do nível de confiabilidade pode ser exemplificada como a esgotamento de energia dos nós. Neste caso, pode ser mais interessante diminuir a confiabilidade da rede para aumentar sua vida útil. Quando o nível de confiabilidade é alterado, um pacote RSP é enviado ao nó origem com o novo nível de confiabilidade solicitado.

Uma vez definido o nível de confiabilidade requerido pela aplicação, os nós da rede podem agir de duas maneiras diferentes, descritas a seguir:

Nível 1 de Confiabilidade

Este nível de confiabilidade visa economia de energia, através da redução de retransmissões, possui baixo custo de *buffers* e aplica-se principalmente a aplicações que possuem alguma redundância de dados ou que possam tolerar perdas.

Depois de receber um pacote de um nó A, o nó B guarda uma cópia do pacote no seu *buffer*, inicia o temporizador, reencaminha o pacote e envia ao nó A um reconhecimento (ACK), passando a ser temporariamente responsável pela entrega do pacote à estação base. Este processo acontece repetidamente, através da rota estipulada pela camada de rede, até que a estação base receba o pacote de dados e envie um ACK ao nó imediatamente anterior. Qualquer um dos nós, ao receber um ACK, pode descartar o pacote enviado, poupando espaço em seu *buffer* conhecidamente reduzido. Esta situação está representada graficamente na Figura 3.2.

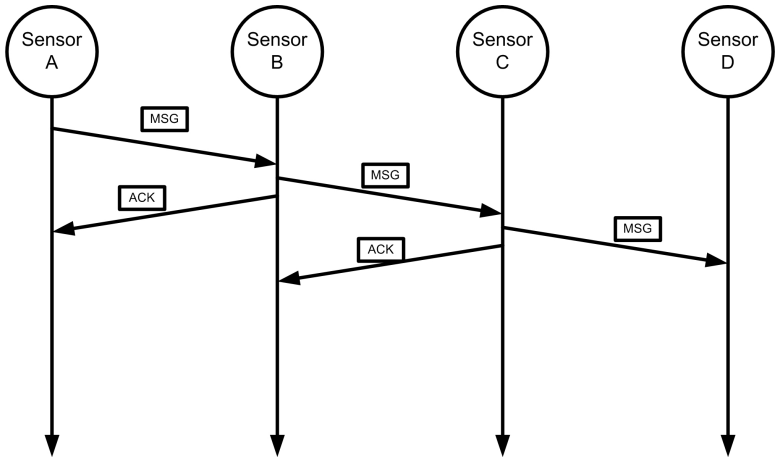

Figura 3.2: O protocolo CTCP com nível 1 de confiabilidade.

Repare que, ao assumir a obrigação de entregar o pacote, o nó intermediário deve manter uma cópia deste pacote em *buffer* até receber o ACK. A ausência de recebimento do ACK gera um esgotamento de temporização e a retransmissão do pacote não reconhecido.

Considere, todavia, a seguinte situação: o nó A, origem do fluxo, envia dados ao nó B. B recebe os dados, armazena no *buffer*, inicia o temporizador, reencaminha o pacote e envia um ACK ao nó A. Por um erro qualquer o pacote não chega ao nó consecutivo (C), mas o nó A recebe o ACK enviado. Logo em seguida, o nó B falha. Deste modo, os dados que estavam sob a responsabilidade do nó B não são retransmitidos para a estação base e o nó A não percebe esta falha. Esta situação só poderá ser minimizada com o aumento do nível de confiabilidade.

Nível 2 de confiabilidade

O algoritmo do nível 2 de confiabilidade aumenta a probabilidade de entrega de uma mensagem, uma vez que a falha de um nó no caminho não interrompe a entrega dos dados.

O nó A envia os dados para o nó B e espera receber o *duplo ACK*. O duplo ACK é gerado da seguinte maneira: B recebe os dados de A e devolve para A o primeiro ACK. O nó B envia os dados para C que devolve para B o primeiro ACK. Quando B receber o primeiro ACK de C envia para A o segundo ACK (duplo ACK). Somente após receber o duplo ACK, A descarta os dados mantidos em *buffer*. Todos os nós

repetem este processo, sucessivamente, até que os dados cheguem à estação base. A estação base deve enviar somente o duplo ACK ao nó imediatamente anterior. A Figura 3.3 representa graficamente esta troca de mensagens.

Se o nó B falhar antes de entregar os dados ao nó C, o nó A não recebe o duplo ACK e retransmite o pacote. Pressupõe-se que as falhas dos nós são monitoradas pelo algoritmo de roteamento e este é responsável por refazer a rota quando da falha de um determinado nó, ou conjunto deles.

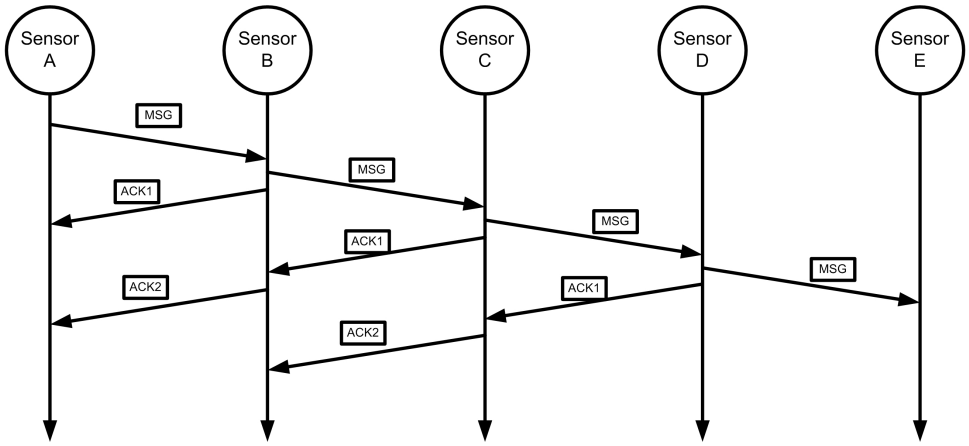

Figura 3.3: O protocolo CTCP com nível 2 de confiabilidade.

Como no nível 1, a ausência de recebimento do primeiro ACK gera um esgotamento de temporização do nó origem e a retransmissão do pacote não reconhecido. Além disso, a ausência de recebimento do duplo ACK, também gera um esgotamento de temporização do nó origem e a conseqüente retransmissão do pacote. Repare que são necessários dois temporizadores distintos (descritos ao final desta seção), uma vez que o tempo de transmissão entre A e B é diferente do tempo entre A e C.

Estimativa do Intervalo de Controle do Temporizador

A exemplo de outros protocolos confiáveis, o CTCP, quando atua com nível 1 de confiabilidade, espera que o nó receptor envie uma confirmação (ACK) toda vez que recebe, com êxito, um novo pacote. Sempre que envia um pacote, o CTCP inicia um temporizador e espera uma confirmação. Se o temporizador expirar antes que os dados do segmento tenham sido confirmados, o CTCP considera que este segmento foi perdido ou danificado e o retransmite.

O protocolo CTCP utiliza um mecanismo de controle de temporização (retrans-

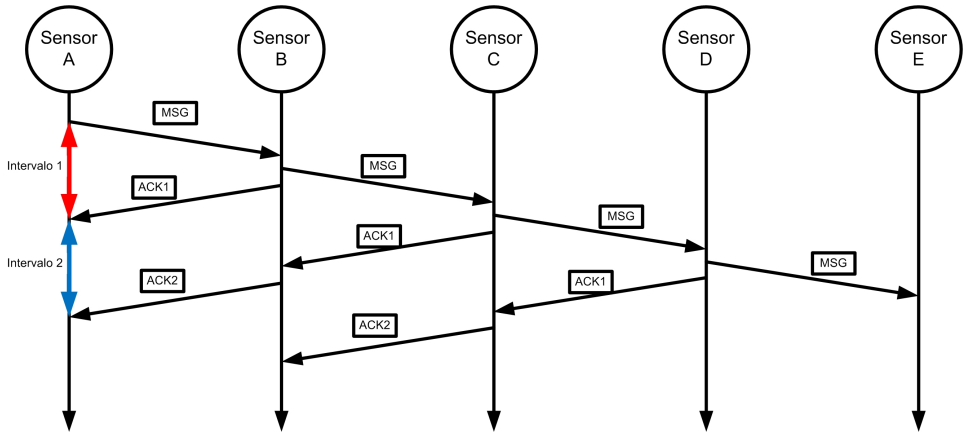

Figura 3.4: Primeiro e segundo intervalos de controle do temporizador.

missão) para recuperar segmentos perdidos. Embora conceitualmente simples, surgem algumas questões sutis quando implementamos um mecanismo de controle de temporização em um protocolo para redes de sensores sem fio.

O intervalo de controle do temporizador (intervalo de tempo medido pelo temporizador) é normalmente definido em função do tempo de transmissão do pacote e de seu reconhecimento (ACK). Na Figura 3.4 estão destacados os dois intervalos de controle que são discutidos nesta seção.

Nas redes em malha sem fio, a métrica Tempo Esperado de Transmissão (*Expected Transmission Time* - ETT) foi proposta por Couto [57]. O ETT é o tempo médio que um pacote de dados precisa para ser entregue com sucesso para o próximo salto. A idéia é calcular periodicamente a taxa de perda de dados e de reconhecimentos positivos (ACKs) para cada vizinho. Esta taxa de perda dos dados é estimada através do envio, em *broadcast*, de quadros com tamanhos semelhantes aos de quadros de dados típicos. Entretanto, cada pacote é enviado em uma das diferentes taxas de envio de dados definidas pelo padrão IEEE 802.11. Já a taxa de perda de reconhecimentos positivos é estimada enviando em *broadcast* pacotes do mesmo tamanho que quadros de ACK na taxa básica da rede. Essas diferenças se justificam, pois os quadros de ACK são normalmente menores que os de dados e o IEEE 802.11 sempre envia quadros de controle na taxa básica da rede [58].

Em uma análise preliminar, conclui-se que o ETT não seria apropriado para as RSSF, pois, nestas redes, existe uma grande restrição energética e todo um esforço deve ser feito no sentido de se poupar energia. Sabe-se que o custo de transmissão

é bem maior do que o custo de processamento. Portanto, a utilização de um mecanismo de estimativa, baseado em *broadcast*, como no ETT, gera um aumento do número de mensagens transmitidas na rede e um desnecessário aumento de consumo.

O intervalo de controle do temporizador, geralmente, deve ser maior que o tempo de transmissão de ida e volta da conexão fim-a-fim (RTT - *Round Trip Time*). Segundo [20], *Round Trip Time* (RTT) é a medida de retardo da transmissão entre dois *hosts*. O tempo de retardo de transmissão (RTT) consiste no total de tempo que um único pacote ou datagrama leva para deixar um equipamento, alcançar o outro e retornar. Na maioria das redes cabeadas de comutação de pacotes, os retardos variam em função do congestionamento. Assim, a medida do tempo de retorno da transmissão é uma média que pode ter desvio padrão alto.

O protocolo CTCP, descrito no início desta seção, trabalha com reconhecimentos positivos, o que possibilita a utilização do RTT. Deve-se lembrar que o RTT é utilizado pelo TCP [48] para medir o tempo transcorrido entre o momento em que o segmento é enviado (para a camada de rede da máquina origem) e o momento em que é recebido um reconhecimento para este segmento (enviado pela máquina destino). A máquina origem e a destino, no TCP, podem estar em redes distintas e distantes. Para utilizar o RTT como base para o intervalo de controle do temporizador do CTCP é necessário considerar cada um dos saltos, ou seja, nesta proposta, o RTT mede o tempo entre a ida de uma mensagem e a volta de um ACK em cada um dos saltos da conexão.

Para reunir dados necessários para um aperfeiçoamento do temporizador, o CTCP registra o instante em que cada segmento é enviado e o instante em que chega uma confirmação para este segmento. A partir dos instantes registrados, o CTCP calcula um período de tempo, chamado amostra do RTT. A Figura 3.5 ilustra o tempo de retorno de 25 pacotes consecutivos. Foi utilizado o nível 1 de confiabilidade do protocolo CTCP.

Na Figura 3.5 pode-se notar que, ao contrário do que acontece nas conexões fim-a-fim, o RTT não possui uma variação absoluta significativa. Os picos registrados no gráfico denotam momentos de retransmissão, onde o tempo medido foi aquele que transcorreu entre a primeira vez que o segmento foi enviado e a chegada do primeiro ACK do referido segmento. A partir destes resultados preliminares, optou-

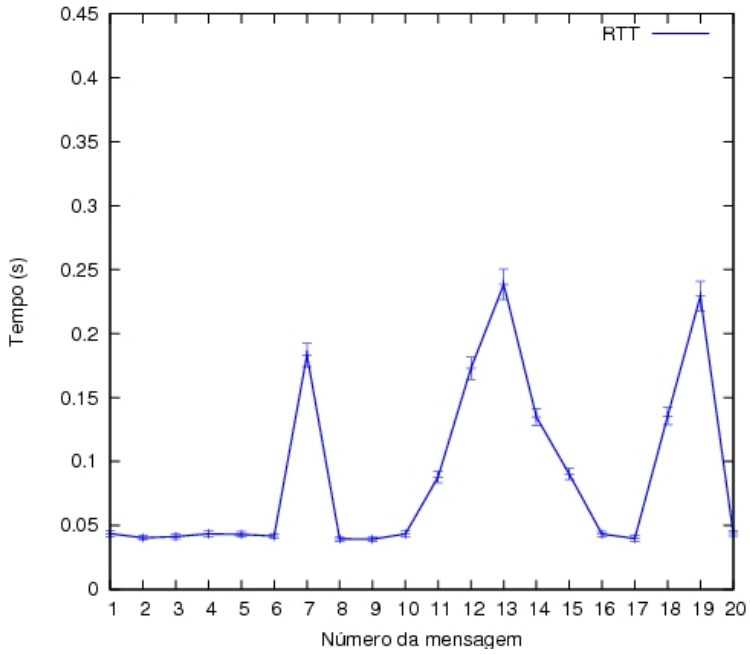

Figura 3.5: Tempo de retorno (RTT) de 25 pacotes consecutivos em uma rede de sensores.

se por basear o mecanismo de temporização adaptável do CTCP no algoritmo de Karn [20, 59]. Este algoritmo, muito conhecido por sua implementação junto ao protocolo TCP, tem sido exaustivamente estudado e calibrado para redes cabeadas e, segundo Comer [20], a experiência prova que o algoritmo de Karn funciona bem, mesmo em redes com alto nível de perdas de pacotes. Este trabalho, pretende estipular os parâmetros α, β e γ (descritos a seguir), para o seu funcionamento nas redes de sensores sem fio.

O CTCP mantém uma média, denominada *RTTPrev*, dos valores de RTT calculados para cada segmento transmitido. Ao obter uma nova medida do RTT, o CTCP atualiza o *RTTPrev* de acordo com a seguinte fórmula:

$$RTTPrev = (1 - \alpha) \times RTTPrev + (\alpha \times Amostra) \tag{3.1}$$

Note que *RTTPrev* é uma média ponderada dos valores de RTT, onde o peso atribuído a uma amostra varia de acordo com o valor estipulado para α. α é uma constante de peso, $0 \leq \alpha \leq 1$, utilizada para avaliar a média antiga em relação à última amostra do RTT. Escolher um valor de α próximo a zero torna a média ponderada imune às alterações de curta duração (por exemplo, um segmento único

que encontra um intervalo longo). Escolher um valor próximo a um faz com que a média ponderada reaja muito rapidamente às alterações dos intervalos. Uma vez determinado o valor adequado de α, o CTCP, ao enviar um pacote, calcula um valor para o intervalo de controle (*Timeout*) como uma função do *RTTPrev*.

$$Timeout = \beta \times RTTPrev \tag{3.2}$$

onde β é um fator constante de peso, $\beta > 1$, que torna o *Timeout* maior que a estimativa atual do tempo de ida e volta. Com o objetivo de detectar um pacote perdido rapidamente, o valor do *Timeout* deve ser próximo ao RTTPrev e β deve possuir valor próximo de um. A rapidez na detecção de um pacote perdido aumenta a vazão da rede, pois o CTCP não vai esperar um tempo longo desnecessário para retransmitir o pacote. Por outro lado, se $\beta = 1$, um pequeno atraso do ACK provoca uma retransmissão desnecessária que consome energia da rede. A especificação deste parâmetro deve considerar um compromisso entre vazão e consumo de energia.

Os valores de α e β foram determinados para redes cabeadas que utilizam o protocolo TCP. Em redes de sensores, estes estudos não foram feitos, até então. Um dos objetivos desta seção é determinar valores adequados para estes parâmetros, considerando o protocolo proposto (CTCP) e a natureza restritiva das redes de sensores.

Na versão inicial da implementação do protocolo CTCP, descrita no trabalho [60], considerou-se um RTT fixo e árbitrário. Contudo, para determinar os valores de α e β, novas simulações foram realizadas com α variando entre $0,1$ e $0,9$. Para β, dois valores foram considerados $1,1$ e $1,3$. Assim, a Tabela 3.1 lista todas as combinações testadas.

Durante a execução das simulações foram criados arquivos de *log* que registravam o tempo real de ida e volta de uma determinada mensagem. A cada um destes tempos foi dado o nome de amostra. A partir desta amostra, o CTCP calcula, baseado nas equações 3.1 e 3.2, o valor a ser atribuído para o próximo intervalo de tempo (previsão). Desta maneira, para cada arranjo de α e β (Tabela 3.1), foram registradas as amostras e previsões do intervalo de tempo para cada mensagem, que totalizavam 24. Para cada arranjo de α e β, calculou-se a média aritmética das amostras e também das previsões. O resultado está na Figura 3.6.

Arranjo	α	β
1	0,9	1,1
2	0,7	1,1
3	0,5	1,1
4	0,3	1,1
5	0,1	1,1
6	0,9	1,3
7	0,7	1,3
8	0,5	1,3
9	0,3	1,3
10	0,1	1,3

Tabela 3.1: Valores de α e β utilizados nas simulações.

O arranjo ideal é aquele onde a diferença entre a previsão (P) e a amostra (A) é a menor possível, guardada uma distância mínima ϵ. Logo, a acuidade do algoritmo proposto é inversamente proporcional à diferença entre previsão e amostra.

Seja $\Delta = P - A$

Deseja-se: $P > A$ e $\Delta \rightarrow (0 + \epsilon)$

Tem-se então as seguintes situações:

- $\Delta < 0$: podem ocorrer retransmissões desnecessárias. O temporizador expirou em P, a mensagem foi retransmitida enquanto a confirmação ainda estava a caminho;

- $\Delta \rightarrow 0$: uma previsão muito próxima das amostras fica susceptível a erros. Daí o uso dos fatores α e β descritos anteriormente;

- $\Delta >>> 0$: dificilmente ocorrem retransmissões desnecessárias. Entretanto, pode haver um aumento da latência, uma vez que há um maior atraso nas retransmissões necessárias.

Na Figura 3.6 pode-se notar que o gráfico é claramente dividido em dois blocos. O primeiro bloco refere-se aos arranjos de 1 a 5 onde $\beta = 1,1$ e o segundo refere-se aos arranjos onde $\beta = 1,3$. Desta maneira, pode-se concluir que o acerto é maior na primeira metade do gráfico, onde $\beta = 1,1$.

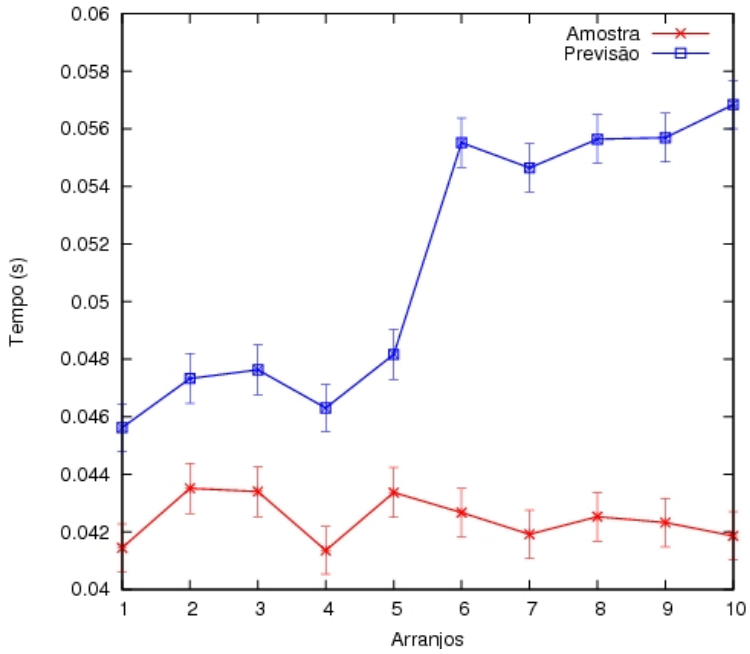

Figura 3.6: Valores das amostras e das previsões em cada um dos arranjos.

Para determinar o valor ideal de α plotamos o gráfico da Figura 3.7. Observa-se que o segundo arranjo, onde $\alpha = 0,7$, possui o menor erro entre previsão e amostra.

O CTCP, assim como o TCP, trabalha com um esquema de confirmação cumulativo, no qual um reconhecimento refere-se a um segmento com número de seqüência único, ou seja, se um segmento com número de seqüência x é enviado, o reconhecimento vem com número de seqüência x. Contudo, se o segmento x for retransmitido várias vezes, quando o reconhecimento chegar, não se pode determinar qual dos segmentos enviados gerou o reconhecimento. Por exemplo, o CTCP forma um segmento e o envia. O temporizador se esgota e o CTCP envia o segmento novamente, em um segundo pacote. Já que os dois pacotes transportam exatamente os mesmos dados e possuem o mesmo número de seqüência, o transmissor não tem como saber se uma confirmação corresponde ao pacote original ou ao retransmitido. Esse fenômeno tem sido denominado ambigüidade de confirmação, e as confirmações do TCP são conhecidas como ambíguas.

Assim, se uma transmissão original e a transmissão mais recente deixam de fornecer tempos de ida e volta precisos, o CTCP não deve atualizar a estimativa do tempo de ida e volta para segmentos retransmitidos. Essa idéia é um dos fundamentos do Algoritmo de Karn [20], que evita o problema de confirmações inteiramente

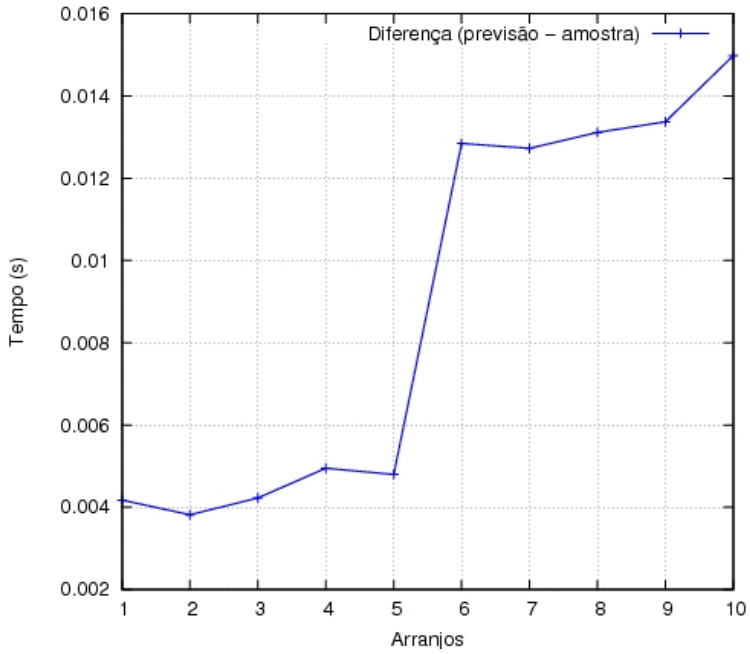

Figura 3.7: Diferença entre P e A em cada um dos arranjos.

ambíguas simplesmente estimando o tempo de ida e volta e ignorando as amostras que correspondam a segmentos retransmitidos. Contudo, Karn ainda sugere a utilização de uma técnica de *backoff* do temporizador, onde o valor do intervalo de controle após um pacote ser retransmitido é aumentado. As implementações usam diferentes técnicas para calcular o *backoff*. A maioria escolhe um fator multiplicativo γ e fixa o novo valor em:

$$Timeout\_1 = \gamma \times Timeout \tag{3.3}$$

O algoritmo de Karn determina que as amostras do RTT não devem considerar os segmentos retransmitidos. Assim, uma nova etapa de simulações se inicia, com o objetivo de determinar os valores apropriados de γ. A Tabela 3.2 lista as combinações de parâmetros utilizados. Os resultados obtidos estão na Figura 3.8.

O valor de γ não influencia no cálculo do RTTPrev, mas sim no *backoff* do temporizador. Contudo, como foi dito anteriormente, só é necessário considerar um *backoff* para o temporizador durante a retransmissão de uma mensagem. Assim, para avaliar a eficiência do fator multiplicativo γ, é preciso avaliar o número de ACKs extras para cada mensagem. Entende-se por ACK extra qualquer ACK duplicado, que chegue ao nó origem. Por exemplo, a mensagem com número de seqüência

Arranjo	α	β	γ	Arranjo	α	β	γ
1	0,9	1,1	1,5	11	0,9	1,1	1
2	0,7	1,1	1,5	12	0,7	1,1	1
3	0,5	1,1	1,5	13	0,5	1,1	1
4	0,3	1,1	1,5	14	0,3	1,1	1
5	0,1	1,1	1,5	15	0,1	1,1	1
6	0,9	1,3	1,5	16	0,9	1,3	1
7	0,7	1,3	1,5	17	0,7	1,3	1
8	0,5	1,3	1,5	18	0,5	1,3	1
9	0,3	1,3	1,5	19	0,3	1,3	1
10	0,1	1,3	1,5	20	0,1	1,3	1

Tabela 3.2: Valores de α, β e γ utilizados nas simulações.

33456 foi enviada do nó origem A para o nó destino B. O nó B deve responder ao nó A com somente um ACK para a mensagem 33456. Entretanto, se o *backoff* do temporizador do nó A estiver mal ajustado, o nó A retransmite para o nó B antes que este possa lhe responder. Em conseqüência, o nó B envia um ACK para cada uma das cópias recebidas e gera os ACKs extras no nó A. Concluindo, um ACK extra denota uma mensagem retransmitida sem necessidade em função de um intervalo de controle mal calculado.

O gráfico da Figura 3.8 está dividido em duas partes. Nos arranjos de 1 a 10, o valor de γ é $1,5$ e nos arranjos de 11 a 20 passou a ter valor 1, ou seja, deixou de influenciar o valor do *Timeout*, uma vez que γ é um fator multiplicativo (veja Equação 3.3).

É importante ressaltar que a ausência do γ, ou seja, $\gamma = 1$, gera grande instabilidade e aumento no número de mensagens retransmitidas desnecessariamente. Durante a análise do número de ACKs extras foi possível perceber que o valor de γ também afeta o número de mensagens entregues ao nó destino. O número de mensagens entregues só volta a crescer para os arranjos 18, 19 e 20 porque o maior valor de β ajudou a compensar a ausência de γ.

Concluindo, após as simulações e medições, assume-se que $\alpha = 0,7$, $\beta = 1,1$ e $\gamma = 1,5$ são os valores adequados para o cálculo do intervalo de controle do protocolo

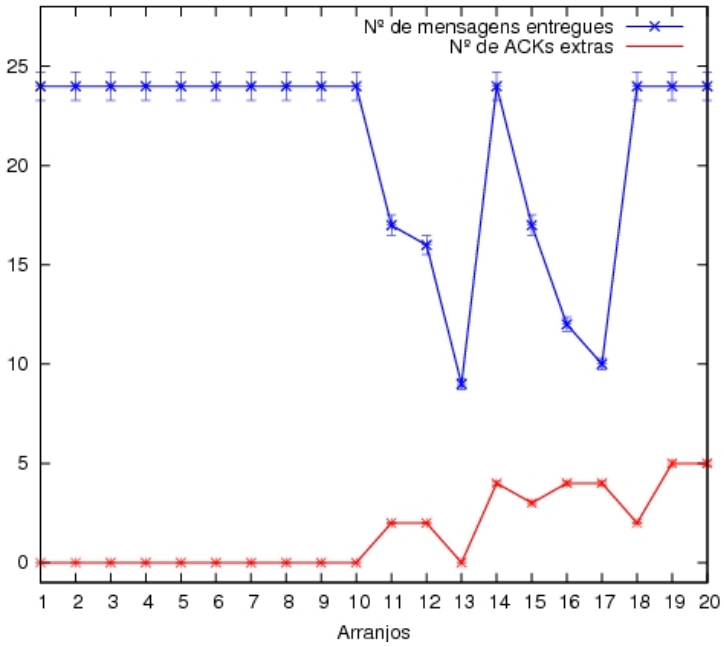

Figura 3.8: Número de mensagens entregues e ACKs extras.

CTCP, em redes de sensores sem fio. Desta maneira, nas simulações descritas no Capítulo 6, são utilizados os valores calculados. Nas simulações que se referem ao nível 2 de confiabilidade é necessário criar um segundo intervalo de controle do temporizador. Este segundo intervalo refere-se ao tempo necessário para esperar o recebimento do ACK duplo (ACK2).

O segundo intervalo de controle utiliza as mesmas equações (3.1), (3.2), (3.3) do primeiro intervalo. Portanto, é preciso, mais uma vez, analisar o valores ideais para α_2, β_2 e γ_2. Realizou-se mais uma bateria de simulações, variando os três parâmetros de acordo com a Tabela 3.2.

Observa-se pela Figura 3.9 que temos vários arranjos com valores reduzidos de Δ. Diferente do que ocorreu no cálculo do $Timeout\_1$, temos uma maior variação nos ACKS repetidos (Figura 3.10). Isto significa que, em muitas situações, ocorrem retransmissões desnecessárias. Assim, dentre o conjunto dos melhores arranjos segundo o valor de Δ (Figura 3.9), identifica-se o arranjo 16 como o mais adequado.

Os parâmetros ideais para o segundo intervalo de controle ($Timeout\_2$), referentes ao arranjo 16, são: $\alpha_2 = 0,9$, $\beta_2 = 1,3$ e $\gamma_2 = 1,0$.

Estes valores são utilizados nas simulações do Capítulo 6.

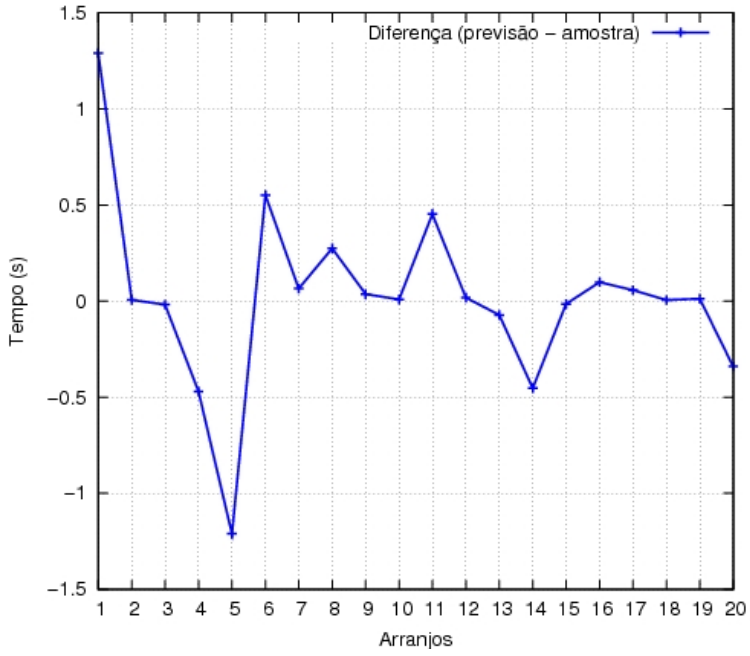

Figura 3.9: Diferença entre P e A em cada um dos arranjos para $Timeout\_2$.

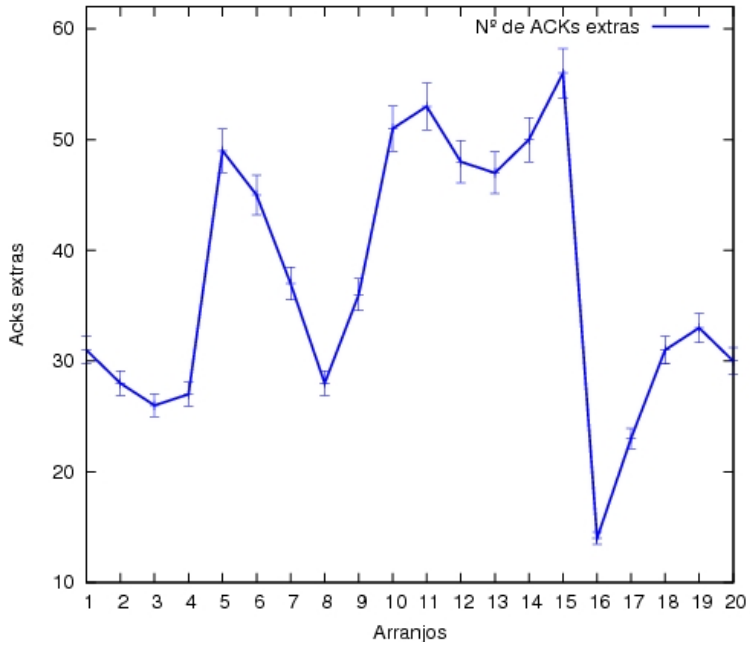

Figura 3.10: Número ACKs extras utilizados na calibragem do $Timeout\_2$.

3.3.3 Detecção e Controle de Congestionamento

Este trabalho pressupõe que cada nó possui um *buffer* de recepção de pacotes. Quando um determinado nó recebe um pacote, ele o coloca no *buffer* de recepção. A aplicação associada a este processo lê os dados a partir deste *buffer*, mas não necessariamente no momento em que são recebidos. Se a aplicação for relativamente lenta na leitura dos dados, ou estiverem chegando pacotes de vários nós ao mesmo tempo, o *buffer* de recepção pode ficar saturado.

Existem alguns protocolos propostos [14, 12] para tratar o problema do congestionamento em RSSF. Cada um possui seus próprios critérios e métodos para detectar e melhorar as situações de congestionamento. Um mecanismo de detecção antecipada e randômico usado em redes tradicionais propõe que um nó intermediário descarte um pacote quando existe congestionamento na rede. A origem percebe o descarte através de um ACK ou NACK. Contudo, descartar pacotes em redes de sensores não é a solução ideal [51], pois, nas redes de sensores sem fio, as perdas de pacotes normalmente se referem a erros de transmissão e não a congestionamentos. Por isso, qualquer perda de pacote inicializa o mecanismo de controle de congestionamento e reduz a taxa de transmissão sem necessidade. Para evitar o descarte de pacotes, este trabalho propõe um algoritmo de controle de congestionamento, composto por três fases, capaz de diferenciar uma perda de pacote relativa a esgotamento de *buffer* de um perda de pacote relativa a erro de transmissão.

A capacidade do CTCP de eliminar as perdas relativas a esgotamento de *buffer* é uma funcionalidade inovadora. Ela não é encontrada, até então, dentre os protocolos analisados.

O controle de fluxo e o controle de congestionamento estão embutidos no mesmo mecanismo. As três fases do algoritmo de detecção e controle de congestionamento estão descritas a seguir.

Primeira Fase: Detecção

Considere uma conexão de múltiplos saltos onde o controle de congestionamento é implementado através da participação de todos os nós intermediários. Cada nó calcula a probabilidade de congestionamento da rede baseado no estado atual de seu próprio *buffer*.

Considere que o nó B, ao receber um pacote, armazena-o no seu *buffer*, até que possa processá-lo. A cada pacote recebido, o algoritmo verifica a ocupação de seu *buffer*. Caso a ocupação seja superior ao patamar T, constata-se a iminência de congestionamento e o algoritmo passa a executar a fase de Realimentação.

Segunda Fase: Realimentação

Todo pacote recebido gera a necessidade de reconhecimento. Assim, quando a ocupação do *buffer* ultrapassa o patamar T, o nó B preenche o campo *Tipo*, pertencente ao cabeçalho do ACK, com o número três (Seção 3.3.4), gerando um pacote denominado STOP. Como o nó A aguarda o recebimento de um reconhecimento (ACK), o nó B utiliza o campo *Tipo* para informar ao nó A que seu *buffer* está com ocupação superior ao patamar T. O nó A, ciente de um congestionamento, reage executando a terceira fase do algoritmo.

Terceira Fase: Reação

Caso a ocupação do *buffer* do nó B esteja acima do patamar T, o nó A interrompe o envio de pacotes a este nó. Tal atitude implica no aumento da ocupação de seu próprio *buffer*, até alcançar o patamar, quando o nó corrente executa a segunda fase do algoritmo, propagando uma redução da taxa de transmissão que pode alcançar toda a rede, dependendo do nível de congestionamento existente no momento.

Cada nó da rede mantém uma tabela com o identificador das conexões ativas e com o ID do vizinho que enviou a última mensagem daquela conexão. A cada mensagem recebida, o endereço do vizinho remetente é atualizado. Através desta tabela é possível saber quais são os vizinhos que estão enviando dados no momento. Caso um destes vizinhos ativos continue a enviar pacotes, pode-se deduzir que o último ACK foi perdido. Um novo ACK é enviado diretamente para ele (*unicast*) com o objetivo de interromper a transmissão. Contudo, os pacotes enviados neste intervalo não são descartados, pois o cálculo do patamar T considera esta situação.

Quando o nó conseguir esvaziar o seu *buffer*, ou seja, quando eles estiverem abaixo do patamar T, um pacote sinalizador (START) é enviado, em *broadcast*, para liberar o encaminhamento de novos pacotes. Um ou mais vizinhos podem não receber o pacote START o que acarreta uma pausa na transmissão do vizinho. A

partir, novamente, da tabela de conexões ativas é possível identificar quais são os vizinhos temporariamente travados e reenviar diretamente para ele (*unicast*) um pacote START. O patamar T é calculado, a princípio, empiricamente, através das simulações realizadas.

Concluindo, toda vez que um nó constata que seu *buffer* está ficando demasiadamente cheio, ele interrompe temporariamente as conexões ativas, até que ele possa processar os pacotes armazenados e liberar o *buffer*. Esta atitude evita que mensagens sejam descartadas por falta de espaço em *buffer*. Desta maneira, pode-se concluir que toda perda de pacotes, é decorrente de erros de transmissão e não de congestionamento.

3.3.4 Formato dos Pacotes do CTCP

O cabeçalho do pacote de dados do CTCP é mostrado na Figura 3.11. Todos os campos são múltiplos de 8 bits em função de restrições relativas à implementação. O primeiro campo do pacote, *ID Conexão*, refere-se ao identificador da conexão e possui tamanho de 8 bits. O campo de nome *NumSeq* que refere-se ao número de seqüência do pacote e possui tamanho de 16 bits. O *Tipo* do pacote identifica o nível de confiabilidade momentânea e possui tamanho de 8 bits. A inclusão deste campo no cabeçalho deu-se em função da necessidade de informar ao nó corrente qual o tipo de confiabilidade utilizada naquele momento. Quando a estação base informar ao nó origem, através do pacote RSP, um novo nível de confiabilidade, o referido campo é alterado nos pacotes que ainda não saíram do nó origem. O campo *EndDestino* possui 16 bits de tamanho e é responsável por armazenar o vizinho para o qual o pacote é enviado, ou seja, o seu *pai* no grafo de roteamento. O *EndOrigem* armazena a origem do pacote e não é alterado durante seu tempo de vida. Por fim, temos o campo *EndIntercept* que armazena o endereço do último nó que interceptou o referido pacote. Nota-se que este último campo é alterado a cada nó pelo qual ele passa. O campo *EndInterceptAnt* armazena o endereço do penúltimo nó que interceptou o referido pacote. Este campo é utilizado para que o nó saiba, mesmo depois de alterar o endereço do *EndIntercept*, para que nó enviar o ACK duplo.

O pacote de dados do CTCP possui ao todo 80 bits de cabeçalho e pode carregar x bits de dados. O valor de x dependerá do MSS (*Maximum Segment Size*) estipulado

para a camada de transporte.

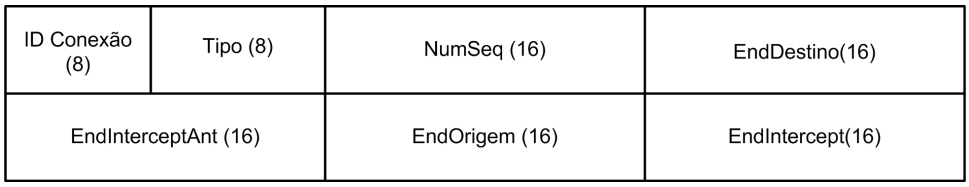

Figura 3.11: Cabeçalho do pacote de dados do CTCP.

O pacote de reconhecimento do CTCP, ver Figura 3.12, foi projetado para ter o menor tamanho possível com o objetivo de diminuir sua probabilidade de perda e o consumo de recursos em geral. O campo *RecNumSeq* contém o número de seqüência da mensagem que está sendo reconhecida. O campo *Tipo* pode possuir vários valores pré-determinados. Seu valor é 1 quando o pacote for um ACK1 e 2 quando o pacote for ACK2. O número 3 indica que o nó está com sua ocupação de *buffer* superior ao patamar (STOP). É importante ressaltar que o pacote de reconhecimento não carrega dados, possuindo, portanto, sempre 24 bits.

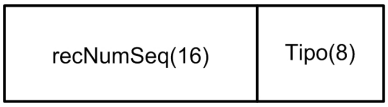

Figura 3.12: Pacote de reconhecimento do CTCP.

O pacote de recuperação de transmissão, chamado START, ver Figura 3.13, possui dois campos. O primeiro chamado *ID Nó*, identifica o nó que está enviando o pacote. O segundo campo possui um identificador único e constante em todos os pacotes *START*.

Figura 3.13: Pacote START do CTCP.

3.4 Especificação Formal do Protocolo CTCP

Os protocolos de comunicação utilizados em redes de computadores e sistemas distribuídos constituem sistemas de alta complexidade. Um protocolo desses

caracteriza-se pela existência de entidades que se comunicam através da troca de mensagens de acordo com procedimentos e formatos pre-estabelecidos.

A engenharia de protocolos aplica-se à concepção, à realização e à análise de protocolos de comunicação utilizados nos mais variados ambientes. Ela oferece orientação segura aos projetistas de protocolos, no que se refere à resolução de problemas de comunicação em ambientes distribuídos. As soluções que ela oferece materializam-se na forma de *software*, *hardware* e documentação. Ela beneficia-se de normas e padrões estabelecidos por organizações nacionais e internacionais, assim como da experiência dos técnicos envolvidos e da habilidade do trabalho em equipe [61].

As ferramentas que podem auxiliar um engenheiro de protocolos podem ser classificadas em dois tipos, segundo [61]: Ferramentas Conceituais e Ferramentas de Uso Prático. As ferramentas conceituais, como as técnicas de descrição formal, os estilos de especificação e as metodologias de desenvolvimento, constituem recursos de projeto que direcionam e dão expressividade às concepções do projetista. As ferramentas de uso prático, como o *software* de edição, análise e transformação de especificações, constituem uma base de apoio que oferece rapidez, segurança e produtividade ao trabalho.

Para a modelagem e análise formal do protocolo CTCP, foram empregados dois métodos, as Redes Predicado Ação [62, 63], uma extensão das Redes de Petri, e o CCS (*Calculus of Communicating Systems* [64]), de Robin Milner [65, 66]. São métodos de especificação formal que permitem o desenvolvimento de sistemas com um mínimo de ambigüidades, através de uma sintaxe e semântica bem definidas. A especificação formal de protocolos possibilita uma análise de comportamento funcional de certas propriedades, como sincronismo, ausência de bloqueios (*deadlocks*) e seqüência correta de mensagens, além da verificação da consistência da especificação.

3.4.1 Entidades, Primitivas e Serviços

As entidades de protocolo constituem os atores, que se comunicam, virtualmente, trocando Unidades de Dados, segundo um conjunto de regras e convenções preestabelecidas. O serviço de comunicação constitui o conjunto de recursos que é oferecido, em ambiente distribuído, às entidades de protocolo, para que essas entidades se co-

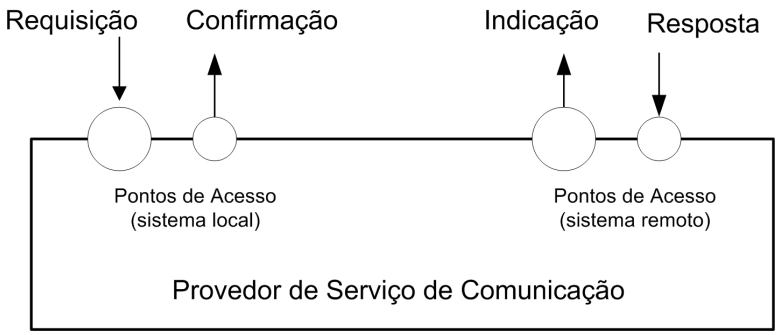

Figura 3.14: Comunicação entre dois nós no nível 1.

muniquem através da troca ordenada de mensagens. O uso de tal serviço - através das primitivas de serviço Requisição, Indicação, Resposta e Confirmação - ocorre em diversos pontos de acesso disponibilizados pelo Provedor de Serviço correspondente.

A Figura 3.14 apresenta as entidades, primitivas e os serviços de comunicação de um protocolo.

A Figura 3.14 representa um sistema distribuído cujo serviço de comunicação oferecido pela camada de transporte permite a comunicação real das entidades de protocolo. Por exemplo, uma entidade de protocolo da Camada de Aplicação comunica-se diretamente com uma entidade par, isto é, uma entidade do mesmo protocolo, através do serviço de comunicação oferecido pela Camada de Transporte desse modelo. A este sistema distribuído dá-se o nome de Provedor de Serviço de Comunicação.

O provedor é, usualmente, representado por uma "caixa-preta", que dispõe de portas de comunicação (locais e remotas) denominadas Pontos de Acesso ao Serviço (PAS). Essas portas são representadas, algumas vezes, por uma "caixa-preta"onde ocorrem as primitivas de serviço. As portas estão localizadas em diferentes locais de um sistema distribuído.

Primitivas de serviço são os eventos que ocorrem nos pontos de acesso aos serviços distribuídos de comunicação. Usualmente, são definidos quatro tipos de primitivas de serviços [61]:

- Requisição - para representar o evento com o qual uma entidade de protocolo realiza a solicitação de algum serviço ao provedor;
- Indicação - para representar o evento no qual o provedor oferece um serviço a uma entidade de protocolo, de sua própria iniciativa ou após a ocorrência de

uma Requisição local ou remota;

- Resposta - para representar o evento no qual uma entidade de protocolo solicita, ao provedor de serviço, que transmita a resposta correspondente a uma requisição que lhe chegou através de uma primitiva Indicação;
- Confirmação - representa o evento correspondente à chegada de uma resposta enviada pela ocorrência de uma primitiva Resposta.

Em cada uma das seções seguintes, são especificadas as entidades, primitivas e serviços do protocolo CTCP. Além disso, é utilizada uma ferramenta, chamada *Edinburgh Concurrency Workbench (CWB)* [67, 68], que serve para a manipulação e análise de sistemas concorrentes. Em particular, o CWB permite a validação da modelagem, equivalência e ordenação de controles através da utilização de diferentes processos semânticos [67].

3.4.2 Especificação Formal do Nível 1 de Confiabilidade

Rede Predicado-Ação

A rede predicado-ação da Figura 3.15 descreve o comportamento de dois sensores que executam o protocolo CTCP com nível 1 de confiabilidade. O nó A representa o transmissor e o nó B o receptor. O temporizador do nó A foi propositadamente omitido para simplificar o entendimento da rede.

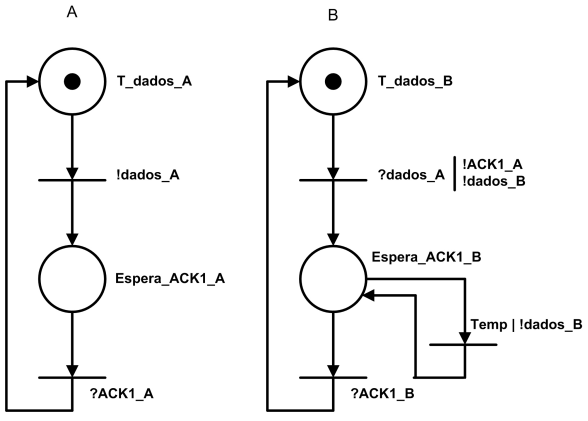

Figura 3.15: Rede Predicado-Ação para a confiabilidade nível 1.

O significado de cada uma das primitivas do protocolo está detalhado na Tabela 3.3

solicita_transp	A camada de aplicação solicita o serviço de transporte
dados_A	Sensor A envia uma mensagem de dados ao Sensor B
dados_B	Sensor B envia uma mensagem de dados a uma entidade externa (Sensor C)
ACK1_A	Sensor B envia um reconhecimento ao Sensor A
ACK1_B	Entidade externa (Sensor C) envia um reconhecimento ao Sensor B
libera_bufA	Confirmação de que o transporte foi executado pelo Sensor A
libera_bufB	Confirmação de que o transporte foi executado pelo Sensor B

Tabela 3.3: Primitivas do nível 1 de confiabilidade do Protocolo CTCP.

Entidades e Serviço do Nível 1

A Figura 3.17 representa um sistema distribuído cujo serviço de comunicação oferecido pelo Nível 1 de confiabilidade do protocolo CTCP permite a comunicação real das entidades de protocolo ilustradas na Figura 3.16. Na notação utilizada, ! denota o envio de um pacote e a ? o recebimento dele.

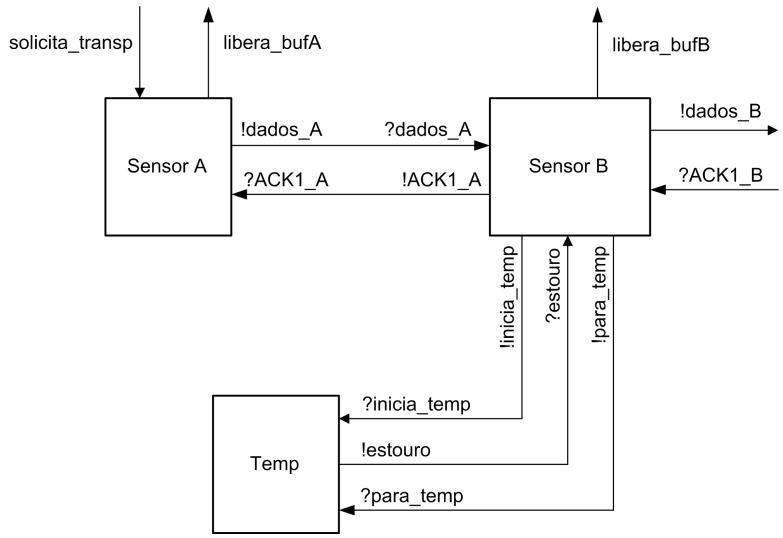

Figura 3.16: Diagrama das entidades do CTCP e mensagens trocadas entre elas.

A Figura 3.16 ilustra as três entidades do protocolo (sensor A, sensor B e temporizador), além da interação entre elas. A primeira seta vertical que aparece na Figura 3.17 possui uma legenda onde se lê *solicita_transp*. Esta seta indica uma Requisição de serviço feita pela camada de aplicação ao CTCP. Ao receber a solicitação da aplicação o nó A processa a informação e envia um pacote de dados (dados_A) ao nó B. O pacote recebido é processado e enviado ao nó C, desta vez com nome de dados_B. Imediatamente após o envio de dados_B, o nó B envia o

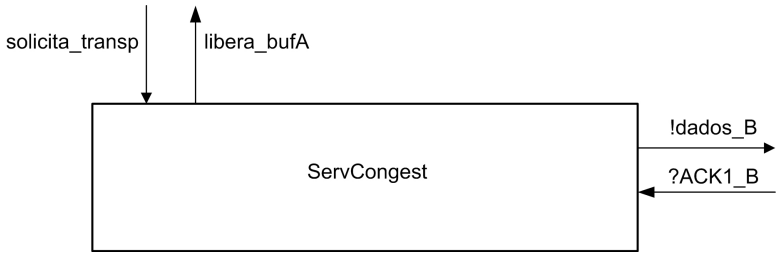

Figura 3.17: Serviço fornecido pela confiabilidade nível 1 do CTCP.

ACK_A para o nó A. Ao receber ACK_A o nó A está pronto para liberar sua área de *buffer*(libera_bufA). O nó B vai interagir com o temporizador para fazer (ou não) retransmissões de dados_B até receber o ACK_B esperado. Neste instante, o nó B está apto a liberar seu próprio *buffer*(libera_bufB).

Codificação da Confiabilidade Nível 1 em CCS

A seguir, pode-se ver o algoritmo do nível 1 de confiabilidade formalmente especificado em CCS. Considere que $B!dados\_A$ significa o envio de $dados\_A$ ao nó B, e $A?dados\_A$ significa a recepção de $dados\_A$ enviado por A. Um maior detalhamento sobre CCS pode ser encontrado em [64].

$Nivel\_1 = A|||B|||Temporizador$
$A = solicita\_transp.!dados\_A.Espera\_ACK1\_A;$
$Espera\_ACK1\_A = ?ACK1\_A.!libera\_bufA.A;$
$B = ?dados\_A.!ACK1\_A.!dados\_B.!inicia\_temp.Espera\_ACK1\_B;$
$Espera\_ACK1\_B = ?ACK1\_B.!para\_temp.!libera\_bufB.B + ?estouro\_temp.!dados\_B.!inicia\_temp.Espera\_ACK1\_B;$
$Temporizador = ?inicia\_temp.(!estouro\_temp.Temporizador + ?para\_temp.Temporizador);$

Resultado da Simulação no CWB

Os resultados da simulação do Nível 1 de confiabilidade do protocolo CTCP, demonstram que o protocolo atende às propriedades desejadas de ser uma rede limitada, pois o protocolo possui um número finito de estados; de ser reiniciável, pois é possível retornar ao estado inicial a partir de qualquer estado da rede; de ser viva, pois todos os estados são alcançáveis a partir de um estado inicial. A análise demonstrou também que o protocolo não possui laços infinitos ou bloqueios(*deadlocks*).

```
Command: print;
** Agents **
agent A = solicita_transp.'dados_A.Espera_ACK1_A;
agent B = dados_A.'dados_B.'ACK1_A.'inicia_temp.Espera_ACK1_B;
agent Espera_ACK1_A = ACK1_A.'libera_bufA.A;
agent Espera_ACK1_B = ACK1_B.'para_temp.'libera_bufB.B + estouro_temp.'dados_B.'inicia_temp.Espera_ACK1_B;
agent nível1 = (A — B — Temporizador)/Internos;
agent Servnível1 = solicita_transp.'dados_B.'libera_bufA.ack_B.'libera_bufB.Servnível1;
agent Temporizador = inicia_temp.('estouro_temp.Temporizador + para_temp.Temporizador); ** Action sets **
set Internos = ACK1_A,dados_A,estouro_temp,inicia_temp,para_temp;
Command: deadlocks Servnivel1;
None.
Command: deadlocks nivel1;
None.
```

3.4.3 Especificação Formal do Nível 2 de Confiabilidade

Rede Predicado-Ação

A Figura 3.18 ilustra a Rede Predicado-Ação do Nível 2 de confiabilidade do protocolo CTCP. Novamente, o nó A representa o nó transmissor e o nó B o nó receptor. Repare que a especificação do Nível 2 é uma evolução do Nível 1, onde são incluídos alguns estados e trocas de mensagens. Na Tabela 3.4 estão, somente, as primitivas do protocolo que se referem exclusivamente ao Nível 2.

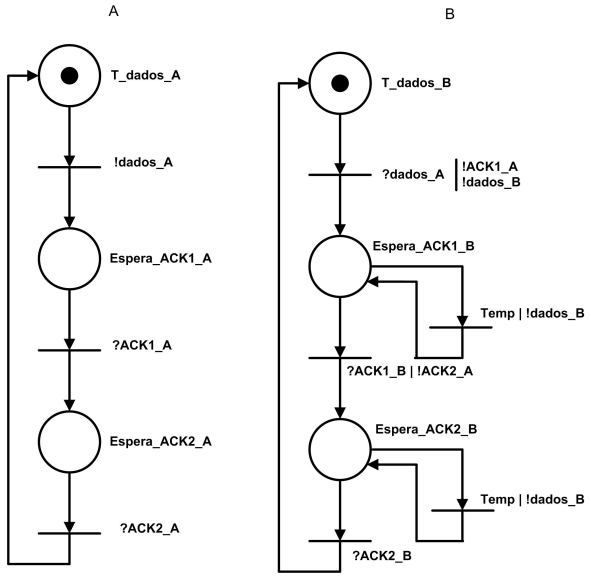

Figura 3.18: Comunicação entre dois nós no nível 2.

ACK2_A	Ao receber ACK1_B o Sensor B envia um segundo reconhecimento ao Sensor A
ACK2_B	Segundo reconhecimento recebido pelo Sensor B

Tabela 3.4: Primitivas exclusivas do nível 2 de confiabilidade do Protocolo CTCP.

Entidades e Serviço do Nível 2

A Figura 3.19 é um diagrama das entidades do protocolo, onde estão em destaque as primitivas do protocolo que se referem exclusivamente ao Nível 2. A Figura 3.20 representa o serviço fornecido pelo Nível 2 de confiabilidade do protocolo CTCP.

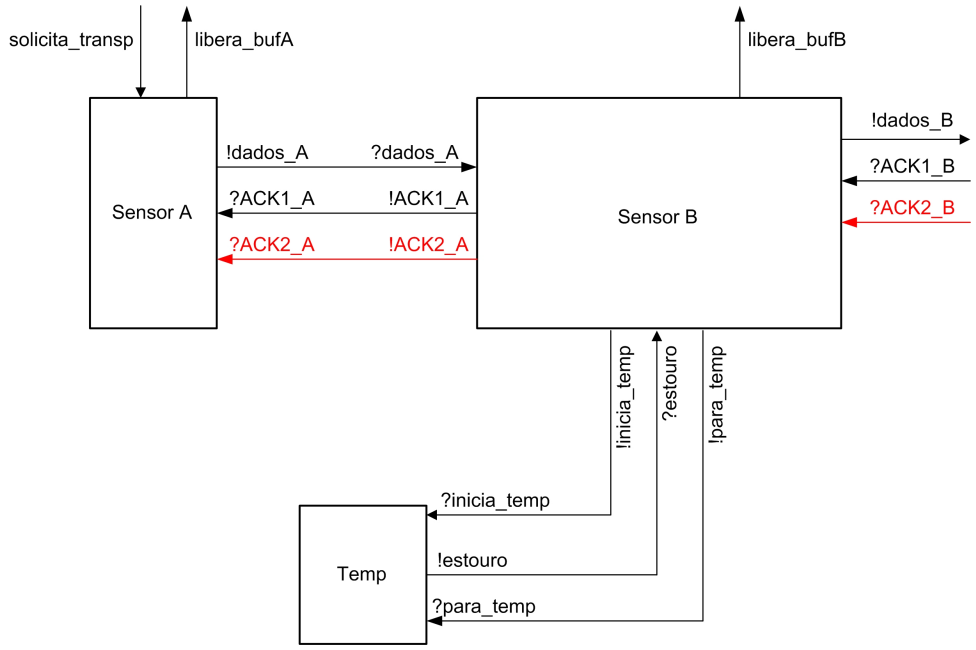

Figura 3.19: Diagrama das entidades do CTCP e mensagens trocadas entre elas.

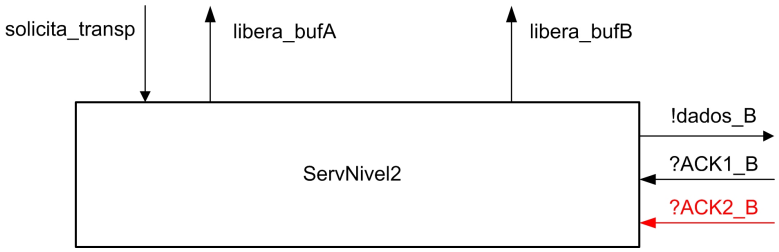

Figura 3.20: Serviço fornecido pela confiabilidade nível 2 do CTCP.

Codificação do Nível 2 de Confiabilidade

A seguir, pode-se ver o algoritmo do nível 2 de confiabilidade formalmente especificado em CCS.

$Nivel\_2 = A|||B|||Temporizador1|||Temporizador2;$
$A = solicita\_transp.!dados\_A.Espera\_ACK1\_A;$
$Espera\_ACK1\_A = ?ACK1\_A.Espera\_ACK2\_A;$
$Espera\_ACK2\_A = ?ACK2\_A.!libera\_bufA.A;$
$B = ?dados\_A.!dados\_B.!ACK1\_A.!inicia\_temp1.Espera\_ACK1\_B;$
$Espera\_ACK1\_B = ?ACK\_B.!para\_temp1.!ACK2\_A.!inicia\_temp2.Espera\_ACK2\_B + ?estouro\_temp1.!dados\_B.!inicia\_temp1.Espera\_ACK1\_B;$
$Espera\_ACK2\_B = ?ACK2\_B.!para\_temp2.B + ?estouro\_temp2.!dados\_B.!inicia\_temp2.Espera\_ACK2\_B;$
$Temporizador1 = ?inicia\_temp1.(!estouro\_temp1.Temporizador1 + ?para\_temp1.Temporizador1);$
$Temporizador2 = ?inicia\_temp2.(!estouro\_temp2.Temporizador2 + ?para\_temp2.Temporizador2);$

Resultado da Simulação no CWB

O resultado apresentado a seguir valida a modelagem do protocolo com Nível 2 de confiabilidade. Ele foi simplificado, com o intuito de diminuir a repetição e facilitar a leitura.

Command: deadlocks Servnivel2;
None.
Command: deadlocks nivel2;
None.

3.4.4 Especificação Formal - Fase Final

Rede Predicado-Ação

Uma vez especificado o nível 1 e 2 de confiabilidade, torna-se necessário especificar o controle de congestionamento. Nesta seção, o controle de condicionamento será adicionado ao nível 2 de confiabilidade. Fez-se esta escolha para que a especificação ficasse o mais completa possível, pois a implementação do controle de congestionamento sobre o nível 1 de confiabilidade representaria uma simplificação do modelo aqui descrito. As Figuras 3.21 e 3.22 representam o lado remetente e o lado receptor durante o controle de congestionamento.

Na Tabela 3.5 estão, somente, as primitivas do protocolo que se referem exclusivamente ao Controle de Congestionamento.

A representação gráfica das entidades do protocolo contidas em sua última etapa de desenvolvimento pode ser encontrada na Figura 3.23.

Codificação do Controle de Congestionamento em CCS

A seguir, pode-se ver o algoritmo do nível 2 de confiabilidade formalmente especificado em CCS. Considere que $B!dados_A$ significa o envio de $dados_A$ ao nó B, e $A?dados_A$ significa a recepção de $dados_A$ enviado por A.

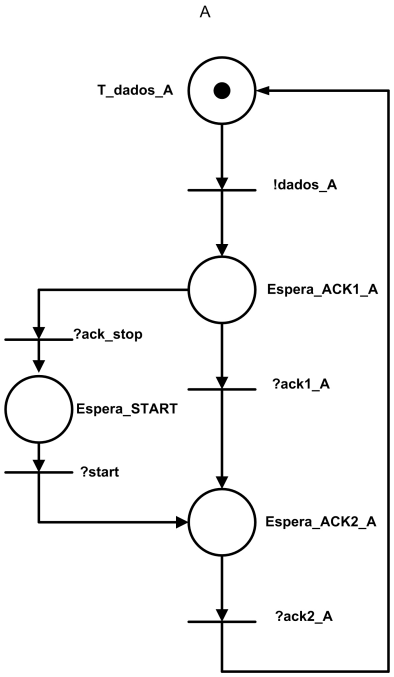

Figura 3.21: Transmissor do controle de congestionamento.

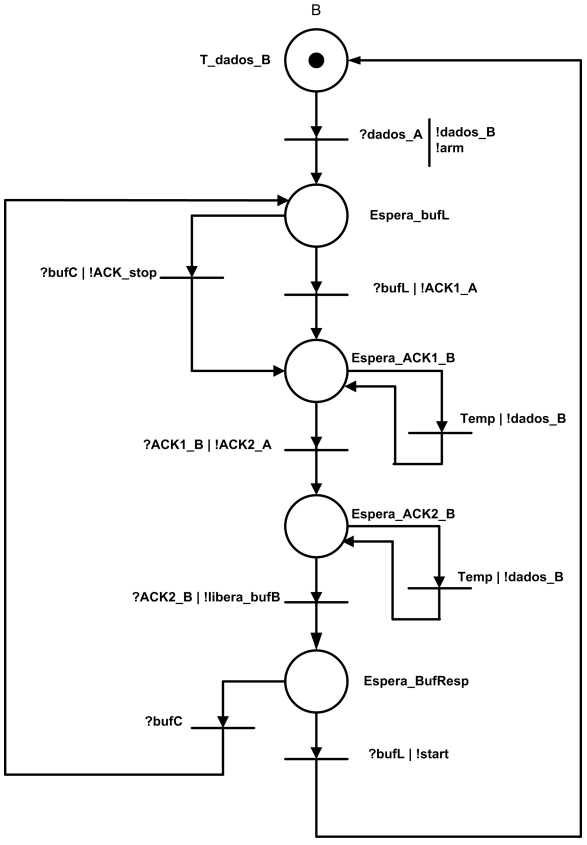

Figura 3.22: Receptor do controle de congestionamento.

ACK_stop	ACK do tipo 3 que interrompe a transmissão de pacotes
start	Libera a transmissão de pacotes
lib_buf	Sensor B informa ao *Buffer* que uma determinada mensagem pode ser apagada
arm	Sensor B solicita informação sobre estado do *Buffer*
bufL	Indica *Buffer* livre (abaixo do patamat T)
bufV	Indica *Buffer* cheio (acima do patamat T)

Tabela 3.5: Primitivas do Controle de Congestionamento do Protocolo CTCP.

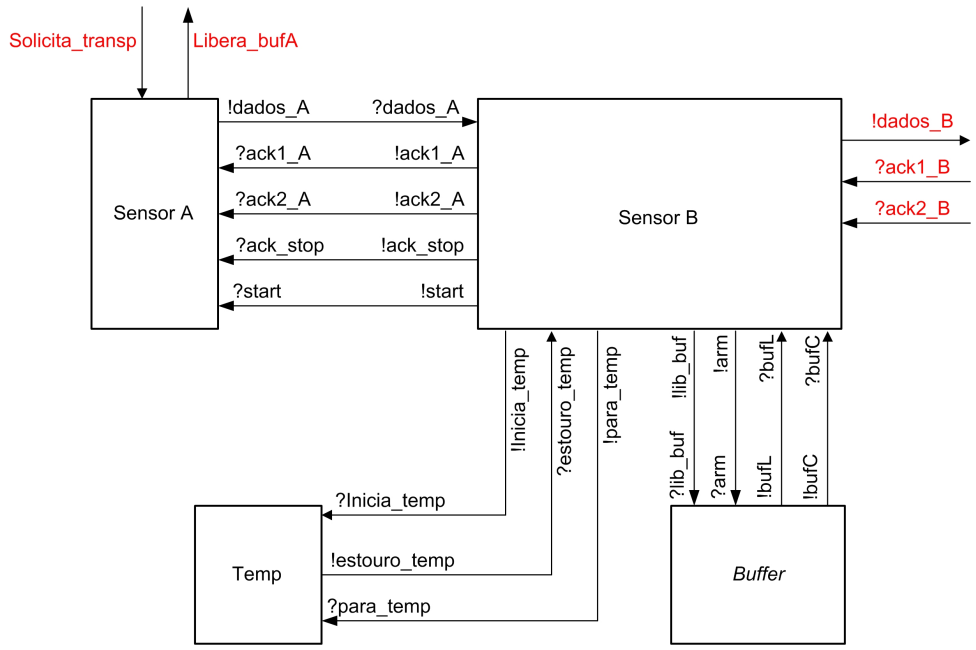

Figura 3.23: Entidades do Controle de Congestionamento do Protocolo CTCP.

$Congest\_1 = A||B||Temporizador1||Temporizador2||Buffer$
$Internos = \{dados\_A, ACK1\_A, ACK2\_A, inicia\_temp1, inicia\_temp2, para\_temp1, para\_temp2, estouro\_temp1, estouro\_temp2,$
$ack\_stop, start, arm, bufL, bufC, libera\_bufB\}$;
$A = ?solicita\_transp.!dados\_A.Espera\_ack1\_A$;
$Espera\_ACK1\_A = ?ACK1\_A.Espera\_ACK2\_A + ?ACK\_stop.Espera\_start$;
$Espera\_ACK2\_A = ?ACK2\_A.A$;
$Espera\_start = ?start.Espera\_ACK2\_A$;
$B = dados\_A.!dados\_B.!inicia\_temp1.!arm.Espera\_bufL$;
$Espera\_bufL = ?bufL.!ACK1\_A.Espera\_ACK1\_B + ?bufC.!ACK\_stop.Espera\_ACK1\_B$;
$Espera\_ACK1\_B = ?ACK1\_B.!para\_temp1.!ACK2\_A.!inicia\_temp2.Espera\_ACK2\_B + ?estouro\_temp1.!dados\_B.!inicia\_temp1.Espera\_ACK1\_B$;
$Espera\_ACK2\_B = ?ACK2\_B.!para\_temp2.!libera\_bufB.Espera\_bufResp + ?estouro\_temp2.!dados\_B.!inicia\_temp2.Espera\_ACK2\_B$;
$Espera\_bufResp = ?bufL.!start.B + ?bufC.Espera\_bufL$;
$Buffer = (arm.(!bufL.Buffer + !bufC.Buffer)) + (libera\_bufB.(!bufL.Buffer + !bufC.Buffer))$;
$Temporizador1 = inicia\_temp1.(!estouro\_temp1.Temporizador1 + para\_temp1.Temporizador1)$;
$Temporizador2 = inicia\_temp2.(!estouro\_temp2.Temporizador2 + para\_temp2.Temporizador2)$;
$ServCongest = solicita\_transp.!dados\_B.ACK1\_B.!libera\_bufA.ACK2\_B.ServCongest$;

Resultado da Simulação no CWB

O resultado apresentado abaixo conclui a validação do protocolo, uma vez que a especificação formal deste protocolo deu-se através de refinamentos sucessivos. Iniciou-se com a especificação do primeiro nível de confiabilidade, o mais simples. Passou-se para o nível 2 de confiabilidade, o que inseriu um pouco mais de complexidade na especificação. Por fim, chegamos ao algoritmo de detecção e controle de congestionamento, que é executado em conjunto com o algoritmo distribuído de confiabilidade dinâmica. O resultado da simulação no CWB, demonstra que o protocolo possui viabilidade, não possui repetições infinitas nem travamentos.

```
Command: deadlocks Servnivel2;
None.
Command: deadlocks nivel2;
None.
```

3.5 Resumo

Neste capítulo foi descrito o funcionamento do protocolo proposto CTCP. Iniciou-se pela abertura e fechamento da conexão. Através de mecanismos de sinalização já explorados pela literatura, foi possível compor uma abertura de conexão, simples e rápida. Em seguida, foi especificado o algoritmo distribuído de confiabilidade dinâmica, proposta até então inexistente na literatura. Este algoritmo possibilita a adequação do CTCP às necessidades da camada de aplicação sem deixar de considerar a otimização do consumo de energia. A Seção 3.3.3 descreve mais um mecanismo inovador desta proposta, o mecanismo de detecção e controle de congestionamento. Ele é capaz de distribuir o acúmulo de pacotes em torno da estação base, por toda a rede e com isso evitar descarte de pacotes dos nós congestionados. A última parte do capítulo descreve a especificação formal do protocolo e possibilita a modelagem probabilística do próximo capítulo.

Capítulo 4

Avaliação Probabilística da Confiabilidade

Este capítulo apresenta uma análise probabilística dos níveis 1 e 2 de confiabilidade. Levou-se em conta a probabilidade de falha nos canais de comunicação e dos nós da rede. Foram desenvolvidos dois modelos matemáticos para a probabilidade de entrega de mensagens ao sorvedouro. O primeiro atende ao caso de um único ACK fim-a-fim e o segundo trata a confiabilidade nível 1. Foi desenvolvido um simulador equivalente aos dois modelos citados e este foi estendido para avaliar também o nível 2 de confiabilidade. Os modelos desenvolvidos a seguir se baseiam, mas não se limitam, àqueles discutidos por Stann *et al.* [16], Spiegel [69, 70] e Lipschutz [71].

4.1 Definições

São utilizados neste capítulo dois termos importantes: a **taxa de sucesso** e **taxa de entrega** dos protocolos. O primeiro termo diz respeito à entrega da mensagem e do respectivo reconhecimento (ACK). O segundo refere-se às mensagens entregues ao nó sorvedouro, mesmo que reconhecimentos tenham sido perdidos. O segundo é sempre maior ou igual ao primeiro.

Os modelos probabilísticos sempre se referem à taxa de sucesso. Isto porque as equações incorporam as probabilidades de execução completa do protocolo.

O simulador permite a avaliação dos dois termos: a taxa de sucesso e a taxa de entrega. Isto porque resultados são armazenados de modo que se identifique a

falha definitiva do protocolo, resultando em não entrega, ou a falha no envio de reconhecimentos, mas com continuidade da propagação da mensagem. Esta última situação pode ser exemplificada como a última tentativa no envio de uma mensagem, o sucesso neste envio e a subseqüente perda do respectivo ACK. Neste caso, houve falha na execução do protocolo, mas a mensagem continuou sendo propagada.

4.2 Probabilidade de Entrega com ACK Fim-a-Fim

As redes tradicionais garantem confiabilidade na camada de transporte através de mecanismos de recuperação de erros fim-a-fim. Esta abordagem não é ideal para as redes de sensores. Ao contrário das redes tradicionais, onde os nós intermediários são apenas roteadores, nas redes de sensores, eles possuem a camada de transporte, o que nos permite distribuir a tarefa de recuperação de erros. Esta colaboração entre os nós na camada de transporte torna-se factível uma vez que todos os nós pertencem à mesma entidade administrativa e podem ser alocados para atingir um mesmo objetivo.

Além de todas as diferenças no modelo de comunicação e serviços das redes de sensores, o maior problema com a recuperação de erros fim-a-fim é a baixa qualidade dos enlaces sem fio. Os sensores, normalmente, trabalham com rádios de baixo alcance, em ambientes com muitos obstáculos e usam técnicas de encaminhamento de mensagens de múltiplos saltos. Desta forma, os erros se acumulam exponencialmente através destes múltiplos saltos.

Considere que as chances de trocar uma mensagem com sucesso através de um único salto seja p. Assim, a taxa de erros no canal de comunicação é dada por $(1-p)$. Seja q a probabilidade de sucesso no envio de um ACK por um único salto. O termo f representa a probabilidade de um nó **não** falhar durante um intervalo de tempo. Nas funções de probabilidade e no programa de simulação, este intervalo transcorre do momento em que uma mensagem é recebida até o momento em que a mesma é encaminhada. Seja R o número máximo de retransmissões que podem ser feitas no nível de transporte.

As análises probabilísticas feitas neste capítulo se baseiam em uma seqüência

linear de nós, sem bifurcações ou rotas alternativas (veja Figura 4.1). Considera-se que o nível de roteamento já definiu o caminho entre origem e destino e este está disponível para transportar as mensagens da camada superior. No caso de falha de um nó e posterior retransmissão, considera-se que uma nova rota já se encontra disponível. Uma análise mais ampla, envolvendo uma dinâmica mais real da rede é feita no Capítulo 6.

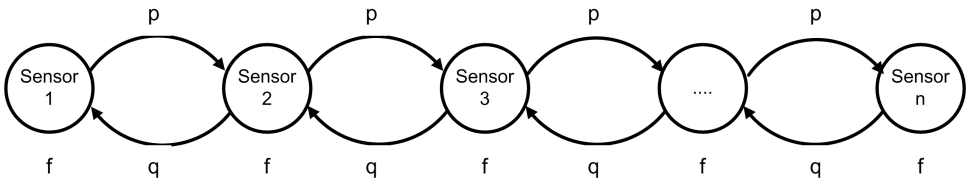

Figura 4.1: Topologia linear.

A utilização de um único ACK fim-a-fim para a recuperação de uma eventual perda está sujeita às vulnerabilidades acumuladas em toda a rota de ida e volta. Assim, para o sucesso na transmissão fim-a-fim, a mensagem tem que atravessar os h saltos da origem (nó fonte) ao destino (nó sorvedouro) e o ACK tem que atravessar os mesmos h saltos de volta (supondo que não houve alteração de rota). O número de nós n é igual a $h + 1$. Em caso de falha, qualquer uma das R retransmissões (novas tentativas) ocorre mais uma vez fim-a-fim. Assim, a probabilidade de sucesso na entrega de uma mensagem, encapsulada em um único pacote, à estação base, h saltos distante do nó origem, é:

$P(S)_{MSG} = p^h$: probabilidade de sucesso em todo o caminho de ida (mensagem);

$P(S)_{procMSG} = f^h$: probabilidade de sucesso no processamento nos $n - 1$ primeiros nós, na ida, $(n - 1 = h)$;

$P(S)_{ACK} = q^h$: probabilidade de sucesso em todo o caminho de volta (ACK);

$P(S)_{procACK} = f^h$: probabilidade de sucesso no processamento nos $n-1$ últimos nós, na volta;

$P(S)_{MSGeACK} = p^h\ q^h\ f^{2h}$: sucesso ida e volta (mensagem e ACK);

$P(F) = 1 - P(S) = (1 - p^h\ q^h\ f^{2h})$: probabilidade de alguma falha ao longo de todo o trajeto;

$P(F)_R = (1 - P(S))^{R+1} = (1 - p^h\ q^h\ f^{2h})^{R+1}$: probabilidade de falha em todas as $R + 1$ transmissões (a original e as R retransmissões).

Assim, a Equação 4.1 nos dá a probabilidade de sucesso com confirmação (ACK) fim-a-fim, para um protocolo que executa até R retransmissões.

$$P(S)_{ACK\ fim-a-fim} = 1 - P(F)_R = 1 - (1 - p^h\, q^h\, f^{2h})^{R+1} \tag{4.1}$$

4.3 Simulador Auxiliar para as Análises Probabilísticas

O simulador desenvolvido consiste em uma classe escrita na linguagem Java [21]. O programa cria uma topologia linear conforme a Figura 4.1. Os parâmetros de entrada estão na Tabela 4.1.

Parâmetro	**Significado**
Rodadas	Número de rodadas executadas na simulação
h	Número de saltos
p	Probabilidade de entrega com sucesso de uma mensagem através de um enlace sem fio (um salto)
q	Probabilidade de entrega com sucesso de um ACK através de um enlace sem fio (um salto)
f	Probabilidade de sucesso no processamento (funcionamento) de um nó, em um intervalo de tempo
R	Número de retransmissões (em caso de não recebimento do ACK)
Confiabilidade	Confiabilidade 1, 2 ou nenhuma (um único ACK fim-a-fim)

Tabela 4.1: Parâmetros de entrada do simulador auxiliar da análise probabilística.

Para cada transmissão de mensagens (que envolve o parâmetro p) ou de um ACK (relacionada a q), o simulador invoca o método **Math.random()**. Este método retorna um número real de dupla precisão (**double**) no intervalo $[0, 1[$. Os valores gerados são escolhidos pseudorandomicamente com uma distribuição (aproximadamente) uniforme no intervalo $[0, 1[$ [72]. O valor retornado pelo método é diretamente comparado à probabilidade de entrega do enlace (o mesmo se aplica ao f), fazendo com que um evento ocorra ou não, com a probabilidade desejada. A Figura 4.2 exemplifica o uso.

```
if ((float) Math.random() <= p) {
    nos[getId() + 1].setRecebeuMSG(true);
        return true;
    } else {
        return false;
}
```

Figura 4.2: Exemplo da aplicação de probabilidades no simulador.

Qualquer valor gerado pelo método **Math.random()**, menor ou igual a p, equivale à transmissão da mensagem. Os demais equivalem a uma perda ou erro irrecuperável de transmissão. O método gera valores seguindo uma distribuição uniforme, o que produz o resultado esperado.

O número de rodadas das simulações deve ser suficientemente grande para que os resultados médios convirjam para os resultados obtidos das equações. O parâmetro h representa o número de saltos (de enlaces) existentes no caminho entre origem e destino ($h = 4$ na Figura 4.1). O valor de h corresponde ao número de nós menos um ($h = n - 1$). Os termos p e q são os mesmos para todos os enlaces e f o mesmo para todos os nós. Os parâmetros p, q, f e R são os mesmos usados na dedução da Equação 4.1.

4.4 Discussão Comparativa dos Termos p e q

As simulações dos Capítulos 3 e 5 foram feitas no *TinyOS Simulator* (TOSSIM) [24]. Este simulador disponibiliza um modelo de rádio chamado *lossy*. O modelo coloca os nós da rede em um grafo direcionado. Cada aresta (a, b) no grafo significa que o sinal de a pode ser ouvido por b, ou seja, refere-se a um salto (um enlace sem fio). Toda aresta possui um valor no intervalo $[0, 1]$, representando a probabilidade de que um bit transmitido por a seja corrompido (invertido) ao ser recebido por b. Por exemplo, um valor $0,01$ significa que cada bit transmitido tem 1% de chance de ser invertido, enquanto $1,0$ significa que todos os bits são invertidos, e $0,0$ significa transmissão sem erros. Cada bit é tratado independentemente [24].

Especificando os erros no nível de bits, o TOSSIM pode capturar muitas causas de

perdas de pacotes e ruídos em uma rede que utilize o sistema TinyOS [22], incluindo *start symbols* perdidos, dados corrompidos e erros em reconhecimentos (ACKs) [24].

O TOSSIM disponibiliza uma ferramenta escrita em Java, *net.tinyos.sim.LossyBuilder*, para a geração de taxas de perdas a partir de topologias físicas. Esta ferramenta modela as perdas observadas empiricamente em um experimento executado por Woo et al. em uma rede TinyOS [73].

A seguir, temos o desenvolvimento da fórmula para calcular a taxa de erros em pacotes (E_p) a partir de taxas de erros em bits (E_b), para o *mica mote* com SECDED (*Single Error Correct, Double Error Detect*) [24]. Os resultados referem-se a **um enlace**.

A probabilidade de sucesso na transmissão do *start symbol* é dada por:

$S_S = (1 - E_b)^9$

A probabilidade de um pacote de um byte não estar corrompido (zero ou um erro de bit) é:

$S_E = (1 - E_b)^8 + (8 \times E_b \times (1 - E_b)^{12})$

Para um pacote com d *bytes*, temos a seguinte probabilidade de sucesso no envio:

$S_{Ed} = (S_E)^d$

O sucesso do *start symbol* e do pacote é:

$S_p = S_S \times (S_E)^d$

E a probabilidade de erro:

$E_p = 1 - S_p = 1 - (S_S \times (S_E)^d)$

Substituindo S_S e S_{Ed} temos:

$$E_p = 1 - ((1 - E_b)^9 \times ((1 - E_b)^8 + (8 \times E_b \times (1 - E_b)^{12}))^d) \quad (4.2)$$

Os ACKs possuem 64 bits (8 bytes) e as mensagens 208 bits (26 bytes). Estes tamanhos correspondem aos cabeçalhos (Seção 3.3.4) somados aos encapsulamentos das camadas inferiores. Deste modo, as mensagens, por serem maiores, tem maior probabilidade de erros de transmissão dos que os ACKs, para um canal com a mesma taxa de erros (Figura 4.3). Observa-se também a rápida degradação do canal (aumento da taxa de erros para ambos os tipos de pacote) com o aumento da taxa de erros por bit.

A Figura 4.4 foi criada a partir do arquivo de erros gerado pelo *LossyBuilder*

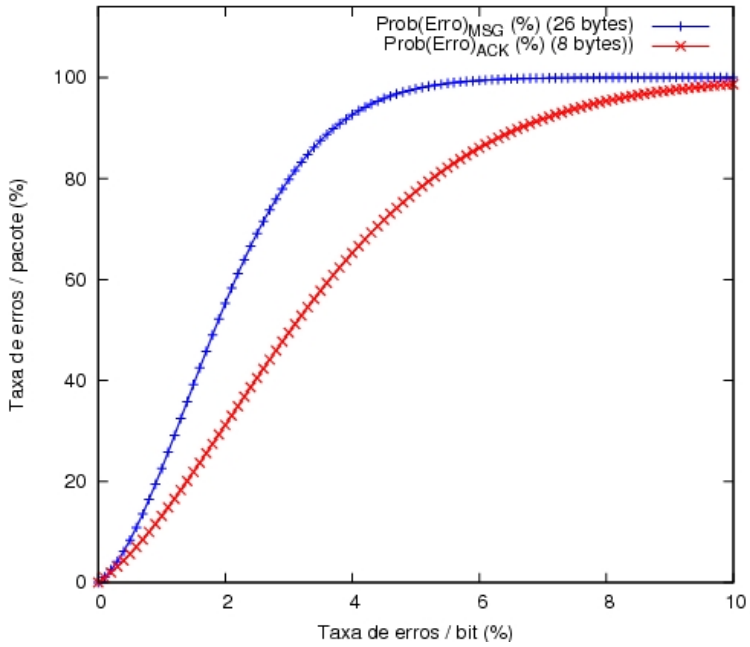

Figura 4.3: Taxa de erros por pacote em função da taxa de erros por bit (um salto).

para a topologia de 100 nós (10 X 10) usada nas simulações dos Capítulos 3 e 5. Nesta figura, nós temos todas as diferentes taxas de erros atribuídas aos diversos enlaces da topologia e as respectivas probabilidades de sucesso (entrega do pacote sem erros) para mensagens e ACKs.

A Tabela 4.2 apresenta a distribuição de probabilidade de entrega de mensagens e ACKs para a topologia de 100 nós utilizada nas simulações e na Figura 4.4. Estas probabilidades referem-se apenas aos nós que estão ao alcance um do outro. Existem vários pares de nós mutuamente inalcançáveis.

Como esperado em redes de sensores sem fio, existem muitos enlaces situados em faixas com baixa probabilidade de entrega, sobretudo para mensagens.

	% de enlaces nesta faixa	
Probabilidade de entrega	**Para mensagens**	**Para ACKs**
75 a 100%	30%	53%
50 a 74,99%	28%	35%
25 a 49,99%	25%	11%
0 a 24,99%	17%	1%

Tabela 4.2: Distribuição de probabilidade de entrega para uma das topologias utilizada.

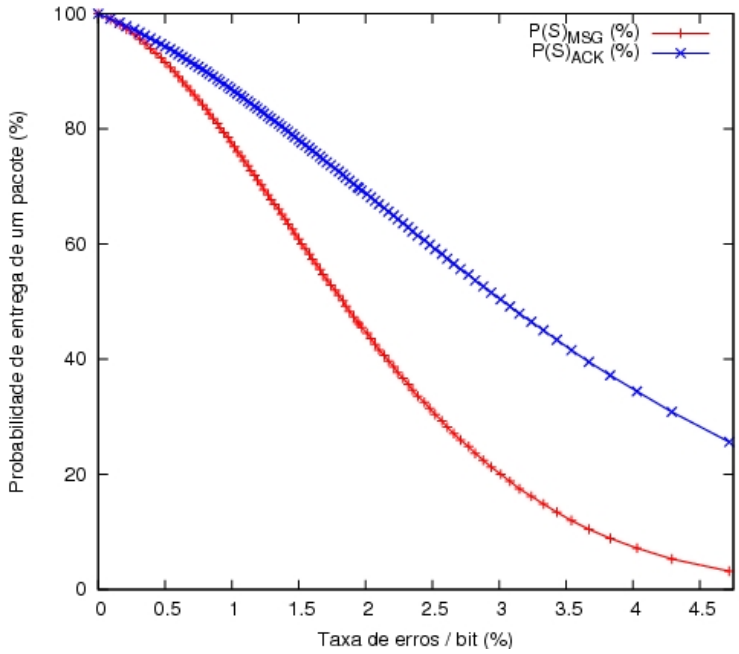

Figura 4.4: Probabilidade de entrega de um pacote em função da taxa de erros por bit (um salto).

4.5 Resultados para ACK Fim-a-Fim

Neste momento, são apresentados dois resultados preliminares: o comportamento de um protocolo que usa confirmação fim-a-fim com a aplicação da Equação 4.1 e com uso do simulador. O objetivo é mostrar o impacto da baixa qualidade dos enlaces sem fio nesta classe de protocolos e validar o funcionamento do simulador frente ao modelo probabilístico apresentado.

Os parâmetros usados foram os seguintes:

- Número de saltos: $3 \leq h \leq 13$;
- Probabilidade de entrega de mensagens: utilizou-se a topologia de 100 nós (10 X 10) citada na Seção 4.4 e outras duas também usadas em simulações (49 nós (7 X 7) e 25 nós (5 X 5)). Após eliminar valores extremos, com altas taxas de erro por bit, tirou-se a média das taxas de erro dos 3204 enlaces da topologia utilizada. Este valor, na Equação 4.2, produziu a taxa de erro média por pacote $\overline{E_p}$. A probabilidade p foi adotada como sendo $1 - \overline{E_p} = 0,63$;
- Probabilidade de entrega de ACKs: com um método análogo ao do item anterior, obteve-se $q = 0,79$;

- Probabilidade de falha do nó: diante dos dois objetivos citados no início desta seção, este parâmetro é menos relevante no momento e é utilizado $f = 1$ (sem falhas de nós);
- Número de rodadas do simulador: 3 execuções com 5×10^3, 5×10^4, 5×10^5, 5×10^6, 5×10^7 e 5×10^8 rodadas em cada uma. As várias faixas mostram a evolução da convergência do simulador em direção ao resultado da fórmula;
- Número de retransmissões: $R = 5$.

A Figura 4.5 confirma o que foi exposto na Seção 4.2. Quando é usado o critério de recuperação com uso do ACK fim-a-fim, temos o acúmulo exponencial dos erros dos enlaces sem fio. Isto causa uma rápida degradação da taxa de entrega.

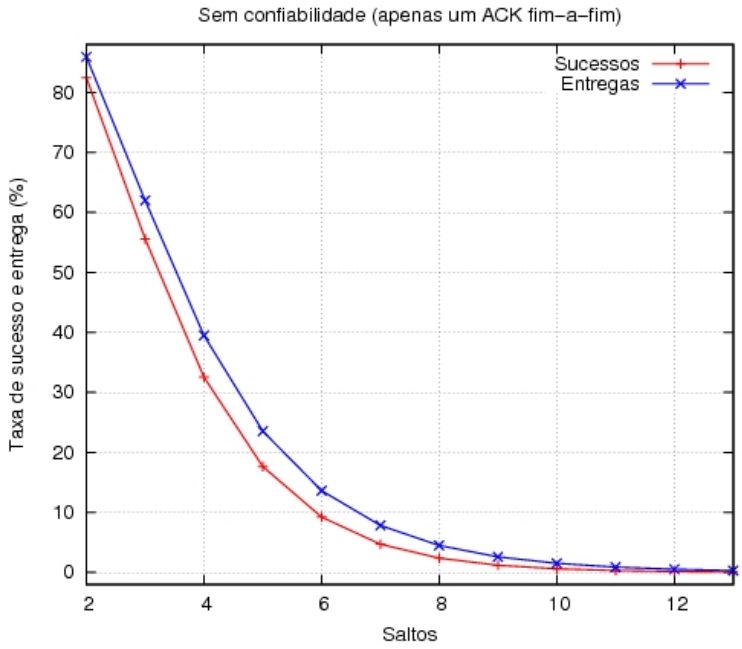

Figura 4.5: Taxa de sucesso e entrega com uso de ACK fim-a-fim.

Nas Figuras 4.6 podemos observar que o simulador reproduz com boa aproximação o comportamento probabilístico modelado na Equação 4.1. A partir de 5×10^5 já se observa um erro médio inferior a $0,5\%$. No decorrer deste capítulo, são realizadas 5×10^7 rodadas em cada simulação, o que leva a erros médios e máximos praticamente nulos (em média $0,0353140\%$ e $0,1337058\%$, respectivamente). Deste modo, sendo o número n de amostras muito elevado, o intervalo de confiança se

torna muito reduzido. Assim, as barras de erro dos gráficos deste capítulo são imperceptíveis.

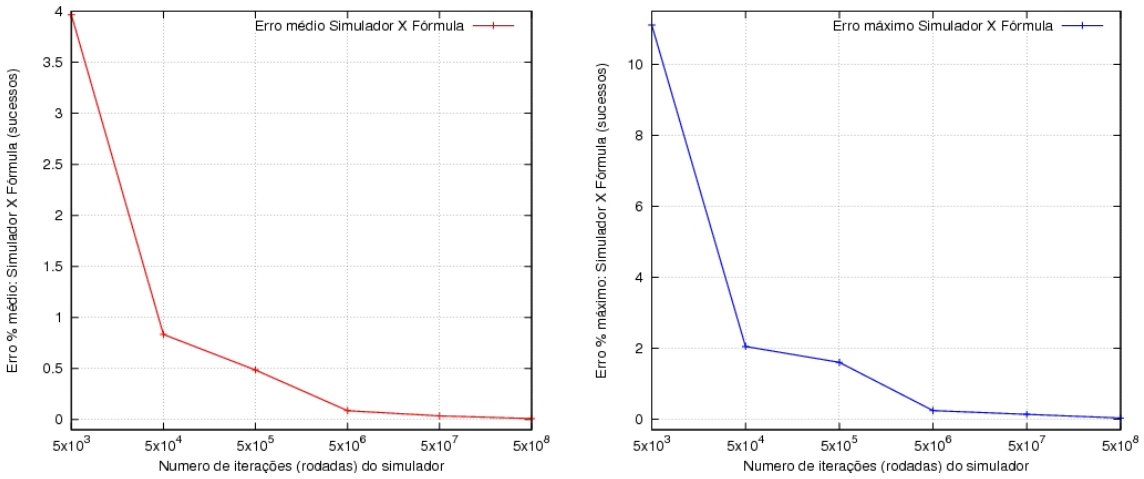

Figura 4.6: Erro médio e máximo do simulador comparado aos valores da Equação 4.1.

4.6 Probabilidade de Entrega para o Nível 1 de Confiabilidade

Conforme descrito no Capítulo 3, o protocolo proposto propõe a recuperação de falhas feita salto a salto, com ACKs enviados por cada vizinho ao nó anterior. Deste modo, a análise probabilística passa a ser a seguinte:

$P(S) = pfq$: probabilidade de sucesso para um único salto (mensagem, processamento e ACK) e uma única tentativa.

Para a confiabilidade nível 1, o parâmetro f tem um significado diferente daquele considerado no caso de um único ACK fim-a-fim. Aqui, ele refere-se à probabilidade de sucesso (funcionamento) durante um **intervalo crítico** do protocolo. Neste intervalo, em caso de falha do nó, a mensagem é definitivamente perdida. Isto pode ocorrer devido ao fato da responsabilidade pelo armazenamento da mensagem ir avançando pela rota até o destino. O envio de um ACK a um nó anterior faz com que este descarte a mensagem correspondente. O nó que enviou este ACK fica responsável pela mensagem até que o próximo lhe envie também um ACK. Estes

intervalos serão detalhados e discutidos na Seção 4.9. No momento, isto nos leva à seguinte probabilidade de falha do protocolo para um salto:

$$P(F)_{1S} = 1 - p \; + \; pf(1-q) \tag{4.3}$$

Observa-se, na Equação 4.3, que as falhas ficam restritas apenas aos enlaces. A referida equação tem o seguinte significado: (*falha no envio da MSG*) **OU** (*sucesso no envio de MSG* **E** *não falha do nó* **E** *falha no envio do ACK*).

Esta forma de modelar o problema nos interessa na medida em que o resultado é aplicado logo em seguida, na dedução da probabilidade de sucesso do protocolo com até R tentativas de retransmissão. Neste caso, uma falha seguida de uma tentativa de retransmissão só faz sentido se a falha se deu nos enlaces (nas transmissões), já que a falha no nó causa a perda definitiva da mensagem e a interrupção da execução do protocolo. Como mencionado, f refere-se ao funcionamento do nó em um intervalo crítico do protocolo. Vale mencionar que este intervalo é muito reduzido.

O sucesso na execução do protocolo pode se dar na transmissão original ou em qualquer uma das R retransmissões tentadas:

$$P(S)_R = pfq + \big(P(F)_{1S} \times pfq\big) + \big(P(F)_{1S} \times P(F)_{1S} \times pfq\big) + \ldots + \big(P(F)_{1S}^{R} \times pfq\big) \tag{4.4}$$

Substituindo $P(F)_{1S}$ temos:

$$\begin{aligned} P(S)_R \;\; = \;\; & pfq + \\ & + \left(\big(1 - p \; + \; pf\,(1-q)\,\big) \times pfq \right) + \\ & + \left(\big(1 - p \; + \; pf\,(1-q)\,\big) \times \big(1 - p \; + \; pf(1-q)\big) \times pfq \right) + \\ & + \cdots + \\ & + \left(\big(1 - p \; + \; pf\,(1-q)\,\big)^{R} \times pfq \right) \end{aligned}$$

O que equivale a:

$$P(S)_R = \sum_{i=0}^{R} \left(\big(1 - p \; + \; pf\,(1-q)\,\big)^{i} \times pfq \right) \tag{4.5}$$

Assim, a probabilidade de sucesso na entrega de uma mensagem, encapsulada em um único pacote, à estação base, h saltos distante do nó origem, é:

$$P(S)_{Conf1} = (P(S)_R)^h = \left(\sum_{i=0}^{R} \left(\left(1 - p \ + \ pf\,(1-q)\,\right)^i \times pfq \right) \right)^h \tag{4.6}$$

4.7 Resultados para o Nível 1 de Confiabilidade

Nesta seção são apresentadas as taxas de sucesso e entrega para o protocolo CTCP executando o nível 1 de confiabilidade.

Foram utilizados os mesmos parâmetros usados na avaliação de um único ACK fim-a-fim (Seção 4.5), com o número de rodadas do simulador fixado em 5×10^7 (3 execuções deste tipo). Em resumo, os parâmetros foram: $p = 0,63$; $f = 1$; $q = 0,79$; $R = 5$. O erro médio do simulador frente aos resultados da Equação 4.6 foi de $0,0030\%$ e o erro máximo de $0,0075\%$. Deste modo, os resultados para as taxas de sucessos são praticamente os mesmos quando extraídos das equações ou do simulador.

Observamos pelas equações 4.1 e 4.6 e pela Figura 4.7 que a recuperação salto a salto oferece uma garantia de entrega muito maior do que o esquema com recuperação fim-a-fim.

4.7.1 Análise do Número de Retransmissões

Quando uma mensagem é enviada e o reconhecimento (ACK) não é recebido, o protocolo faz com que a mesma seja reenviada. A seguir, é apresentada uma análise do ganho ao se aumentar o número R de retransmissões. $R = 0$ significa apenas a transmissão original e nenhuma retransmissão, $R = 1$ corresponde a até uma retransmissão e assim por diante.

As Figuras 4.8 e 4.9 foram extraídas de simulações da confiabilidade 1, feitas com $p = 0,63$, $f = 1$ e $q = 0,79$, como nas anteriores. As análises foram feitas para as instâncias com 20 saltos ($h = 20$). A Figura 4.8 apresenta as taxas de entrega e sucessos para os diversos valores de R e a Figura 4.9 mostra o aumento nestas taxas com relação ao valor anterior, a cada incremento em R. Nesta última figura, foram omitidos os ganhos percentuais para os 3 primeiros valores de R. As taxas de

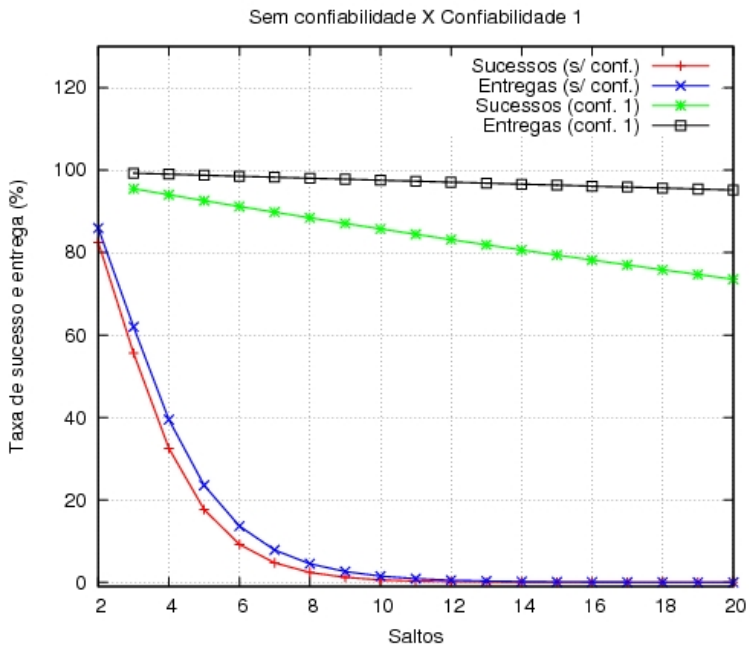

Figura 4.7: Taxa de sucesso e entrega com uso de ACK fim-a-fim e com nível 1 de confiabilidade.

entrega e sucessos são muito reduzidas no início e apresentam uma grande elevação percentual para $R = 1$, 2 e 3, gerando ganhos muito elevados. O maior objetivo é verificar a saturação deste ganho, na busca do R mínimo recomendado.

Observa-se que o uso de, no mínimo, 6 retransmissões é vital para a obtenção de uma alta taxa de entrega para os cenários avaliados. Altas taxas de sucesso do protocolo requerem $R = 9$. As simulações do Capítulo 6 foram feitas com $R = 10$.

4.8 Probabilidade de Entrega para o Nível 2 de Confiabilidade

O nível 2 de confiabilidade executa o protocolo do nível 1 entre um nó e o seguinte, executa novamente o nível 1 entre o segundo e o terceiro e, em caso de sucesso, um ACK2 é enviado do segundo ao primeiro nó (ver Seção 3.3.2). Diante deste fato, e já estando o simulador validado para o nível 1 de confiabilidade, os resultados do nível 2 são extraídos apenas das simulações, segundo os algoritmos 1, 2 e 3.

Já foi visto que o nível de confiabilidade 1 apresenta um desempenho bem superior ao uso de um único ACK fim-a-fim. O protocolo executando o nível 2 de

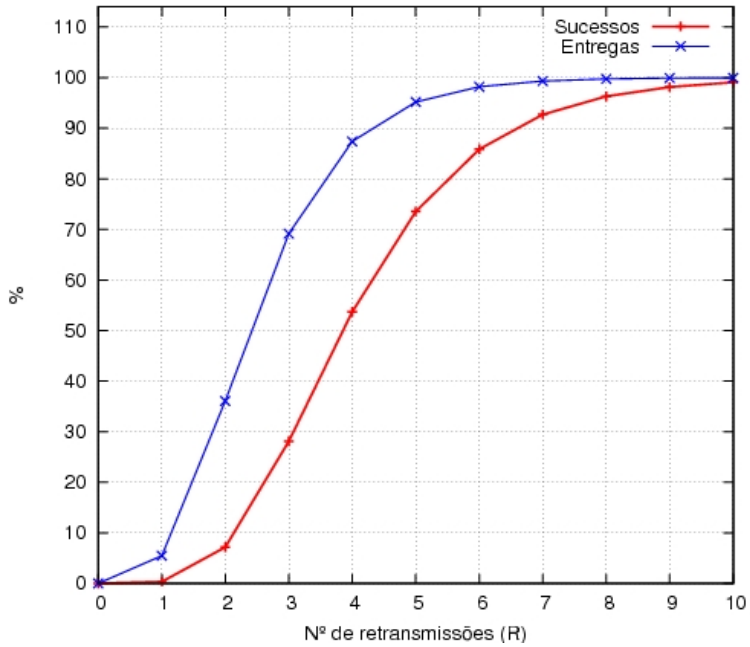

Figura 4.8: Sucessos e entregas: Conf. 1; $h = 20$; R variando de 0 a 10.

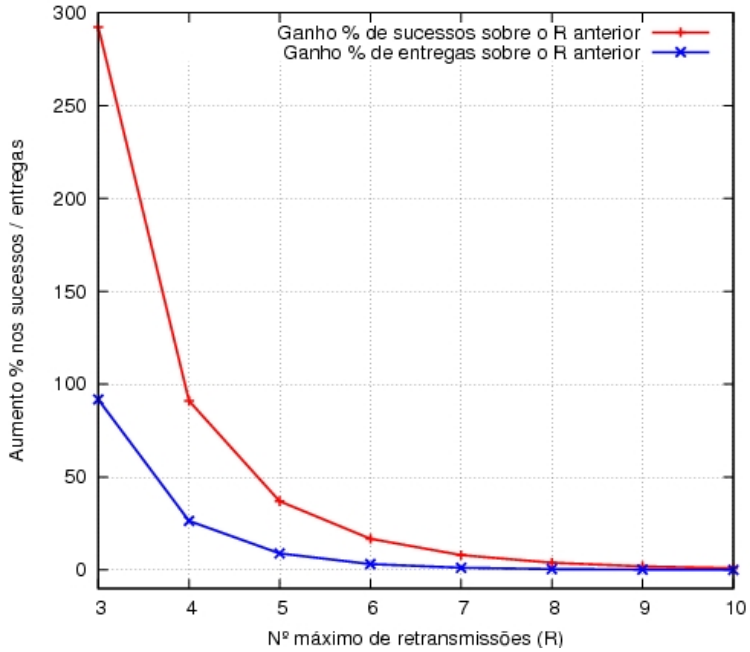

Figura 4.9: Ganho % com o aumento de retransmissões (R de 0 a 10): Conf. 1; $h = 20$.

```
Algoritmo: booleano executaProtocoloConfDois(R, p, q, f, No[] nos)

para todo ( i = 0 ; i ≤ R ; i + + ) faça
    se (nos[ATUAL].executaProtocoloConfUm(0, p, q, f)) então
        se (nos[ATUAL + 1].executaProtocoloConfUm(R, p, q, f)) então
            retorna VERDADEIRO;
retorna FALSO;
```

Algoritmo 1: Nível 2 de confiabilidade.

```
Algoritmo: booleano executaProtocoloConfUm(R, p, q, f, No[] nos)

para todo ( i = 0 ; i ≤ R ; i + + ) faça
    se nos[ATUAL].envia(MSG, p, q)) então
        se Nó não falhou (f) então
            se nos[ATUAL + 1].envia(ACK1, p, q)) então
                retorna VERDADEIRO;
        senão
            retorna FALSO; //falha do nó no intervalo crítico
retorna FALSO; //não entrega ou entrega com falha em algum(ns) ACK(s)
```

Algoritmo 2: Nível 1 de confiabilidade.

```
Algoritmo: booleano envia(tipo, p, q, No[] nos)

caso MSG
    se houve sucesso na transmissão (p) então
        retorna VERDADEIRO;
    senão
        retorna FALSO;
caso ACK1
    se houve sucesso na transmissão (q) então
        se estiver executando Conf 2 então
            se nos[ATUAL -1].envia(ACK2, p, q) então
                retorna VERDADEIRO;
            senão
                retorna FALSO;
        retorna VERDADEIRO;
    senão
        retorna FALSO;
caso ACK2
    se não for o primeiro nó então
        se houve sucesso na transmissão (q) então
            retorna VERDADEIRO;
        senão
            retorna FALSO;
    retorna VERDADEIRO;
```

Algoritmo 3: Envio de mensagens e ACKs.

confiabilidade apresenta taxas de entrega ainda maiores. Assim, são comparados, neste momento, apenas os resultados dos níveis 1 e 2 de confiabilidade.

Foram mantidos os mesmos parâmetros: $p = 0,63$, $f = 1$ e $q = 0,79$, $h = 20$, $R = 5$ e 5×10^7 rodadas do simulador.

Observa-se na Figura 4.10 que o nível 2 de confiabilidade atinge percentuais de entrega ainda maiores, sendo muito pouco afetado pelo número de saltos. A taxa de sucesso do nível 2 é inferior à do nível 1. Esta menor taxa de sucesso se deve a uma maior complexidade do nível 2, requerendo um ACK a mais, o ACK duplo. Ocorre que, em algumas situações, um dos ACKs é perdido, mas a mensagem continua sendo propagada. Entretanto, esta maior complexidade está relacionada à maior robustez do nível 2, que mantém mais cópias redundantes da mensagem, reduzindo a probabilidade de perda definitiva da mesma.

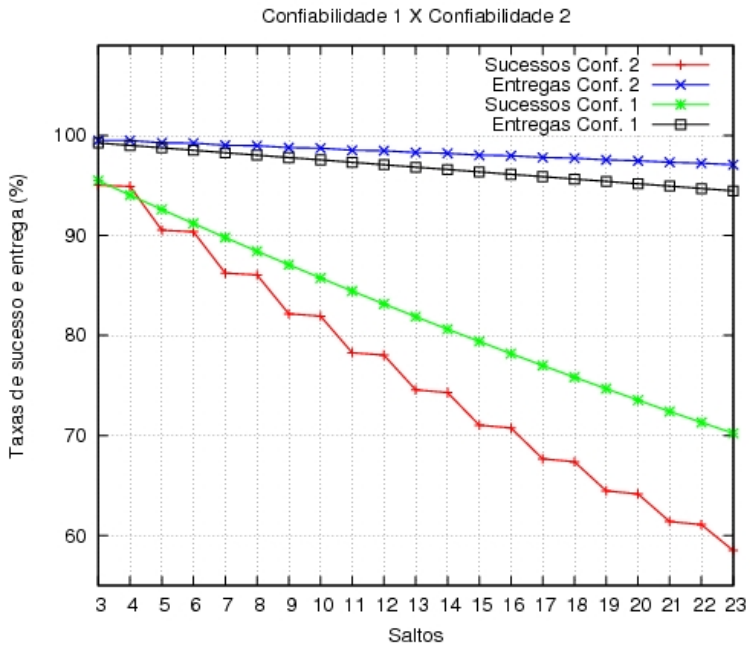

Figura 4.10: Taxa de sucesso e entrega com níveis 1 e 2 de confiabilidade.

4.8.1 Análise do Número de Retransmissões

Como foi feito na Seção 4.7.1 para a confiabilidade 1, nesta seção é avaliado o efeito do número de retransmissões R para a o nível 2 de confiabilidade.

As Figuras 4.11 e 4.12 foram extraídas de simulações da confiabilidade 2, feitas com $p = 0,63$, $f = 1$ e $q = 0,79$, como nas anteriores. As análises foram feitas para

as instâncias com 20 saltos ($h = 20$). A Figura 4.11 apresenta as taxas de entrega e sucessos para os diversos valores de R e a Figura 4.12 mostra o aumento nestas taxas com relação ao valor anterior, a cada incremento em R. Nesta última figura, foram omitidos os ganhos percentuais para os 3 primeiros valores de R pelas mesmas razões citadas na confiabilidade 1.

Mais uma vez, observando as Figuras 4.11 e 4.12 observa-se que o uso de, no mínimo, 6 retransmissões é vital para a obtenção de uma alta taxa de entrega para os cenários avaliados. Altas taxas de sucesso do protocolo, na confiabilidade 2, requerem um valor um pouco maior de possíveis retransmissões, $R = 12$, por exemplo, ou mesmo $R = 10$, já que a partir deste valor o ganho percentual é muito reduzido (Figura 4.12). As simulações da confiabilidade 2, do Capítulo 6, também foram feitas com $R = 10$.

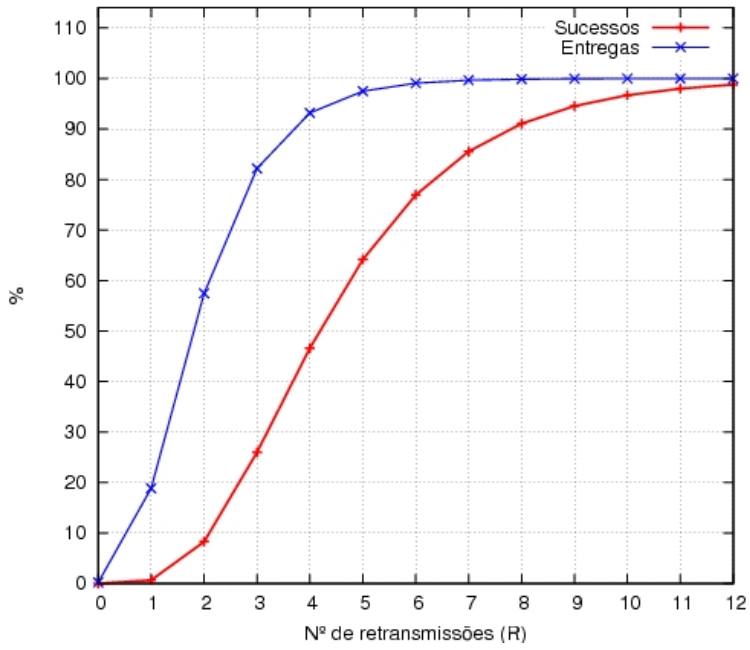

Figura 4.11: Sucessos e entregas: Conf. 2; $h = 20$; R variando de 0 a 12.

4.9 Análise da Falha dos Nós

Como vimos na Seção 4.2, as falhas de comunicação podem ser tratadas através do envio de ACKs e posterior retransmissão corretiva de dados. Um problema adicional reside na possível falha do nó intermediário que contém a informação a ser transmitida.

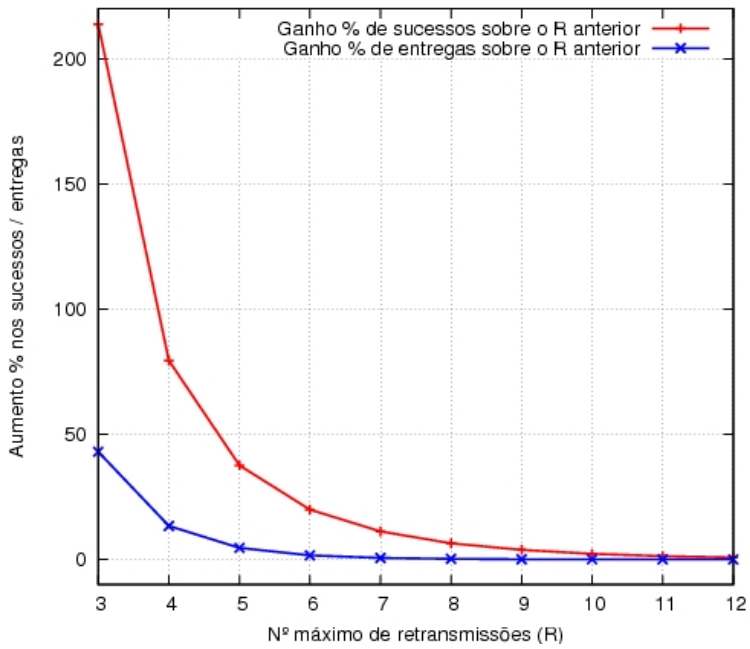

Figura 4.12: Ganho % com o aumento de retransmissões (R de 0 a 12): Conf. 2; $h = 20$.

A confiabilidade de um sistema com processamento distribuído é um importante parâmetro de projeto que pode ser descrito em termos da confiabilidade dos elementos processadores, dos links de comunicação e também da redundância de programas e arquivos de dados.

Segundo Chiang *et al.* [31], nas redes de sensores sem fio, devido à sua distribuição, a confiabilidade e disponibilidade podem ser categorizadas em dois grupos: componentes e processos. A categoria confiabilidade de processos indica a confiabilidade dos componentes. A categoria confiabilidade de processos inclui a interdependência de todos os processos envolvidos, *hardware* e canais de comunicação.

A redundância de *hardware* (nós) tradicionalmente implementada nas redes de sensores aumenta diretamente o nível de confiabilidade dos componentes, mas possui um efeito bem menor no nível da confiabilidade de processos.

No caso da transmissão com ACK fim-a-fim, a informação se mantém armazenada apenas no nó origem, até que um ACK informe que a mesma chegou ao destino. Um defeito, dano ou esgotamento de energia neste nó, antes que a mensagem chegue ao destino, ocasiona sua perda definitiva.

No transporte com nível 1 de confiabilidade, a mensagem se mantém armazenada

sucessivamente nos nós intermediários, até que o nó subseqüentes envie o ACK, assumindo a responsabilidade de mantê-la até que o próximo salto seja concluído. Mais uma vez, uma falha no nó detentor da informação causa a perda definitiva da mesma.

No transporte com nível 2 de confiabilidade, o armazenamento é feito por 2 nós adjacentes ao mesmo tempo, o que reduz a probabilidade de perda da mensagem. Apenas a falha destes dois nós causa a perda da informação.

Foi definido que a probabilidade de um nó **não** falhar é dada por f. A perda total de uma mensagem se dá caso as falhas de nós citadas anteriormente ocorram em intervalos críticos.

Considere os sensores 1, 2 e 3 da Figura 4.1.

O intervalo crítico da confiabilidade nível 1 ocorre quando:

- 2 recebe MSG de 1 com sucesso;
- 2 envia MSG para 3;
- 2 envia ACK para 1 com sucesso;
- 1 descarta MSG;
- MSG de 2 para 3 é perdida;
- 2 falha antes de retransmitir.

Observe ainda que a seqüência $\big(MSG\ (2 \rightarrow 3)\ FALHOU\big)$ **E** $\big(2\ FALHOU\big)$ **E** $\big(ACK\ (2 \rightarrow 1)\ FALHOU\big)$ não implica na perda definitiva da MSG, já que 1 ainda não terá descartado MSG.

No caso da confiabilidade 2, a perda definitiva da mensagem se dá se e somente se os nós 2 e 3 falharem dentro de um mesmo intervalo crítico.

O intervalo crítico da confiabilidade nível 2 ocorre quando:

- 2 recebe MSG de 1 com sucesso;
- 2 envia MSG para 3 com sucesso;
- 2 envia ACK1 para 1 com sucesso;

- 3 envia MSG para 4;
- 3 envia ACK1 para 2 com sucesso;
- 2 envia ACK2 para 1 com sucesso;
- 1 descarta MSG;
- MSG de 3 para 4 é perdida;
- 2 e 3 falham antes de retransmitirem.

Ocorre, entretanto, que a mensagem pode se propagar além de 2 (confiabilidade 1) ou 3 nós (confiabilidade 2). Segundo o protocolo, qualquer nó intermediário que receba uma mensagem a encaminha imediatamente e reconhece, logo em seguida, seu recebimento com um ACK1.

Neste ponto identificamos o conceito de **elasticidade do protocolo**. Uma vez enviada pela origem, MSG se propaga indefinidamente (até o destino no máximo).

Dependendo do sucesso de envio e recebimento dos reconhecimentos (ACKs), há um maior ou menor consumo distribuído de *buffer* para uma mesma MSG. A despeito deste maior consumo, a elasticidade é boa, na medida em que pode gerar mais e mais cópias de MSG (mais uma vez dependendo do sucesso de envio e recebimento dos ACKs), o que reduz a possibilidade de perda definitiva de MSG.

Este conjunto de próximos nós detentores de cópias de uma mesma mensagem (após o detentor mais antigo) formam uma cadeia que é chamada de c.

Assim, generalizando, as falhas críticas citadas anteriormente, ao final dos eventos das confiabilidades 1 e 2, na verdade, se referem à falha de **todos** os nós na cadeia c.

A rigor, basta uma cópia de MSG ainda não reconhecida, ou seja, basta existir um nó que tenha MSG e que ainda não tenha recebido o ACK correspondente, para que MSG ainda não tenha sido perdida em definitivo.

Seja k o grau de elasticidade do protocolo.

k = número de nós subseqüentes ao mais antigo detentor de MSG, que já receberam MSG.

k é o número de elementos da cadeia c.

k varia com todas as caracteríticas momentâneas da rede, acesso ao meio, contenção, condições climáticas, densidade de nós, tráfego, carga de processamento dos nós entre outros.

Como a chance de sucesso no envio de um ACK é maior do que a do envio de uma MSG (Seção 4.4), a maior probabilidade é de termos mais ACKs liberando *buffers* e reduzindo k do que mensagens se propagando além do esperado.

O mais provável, então, é que o protocolo se comporte da maneira esperada, ou seja, com 2 cópias de MSG na rede a cada instante ($k = 1$) para a confiabilidade 1 ou com 3 cópias de MSG na rede ($k = 2$) para a confiabilidade 2.

4.9.1 Análise das Falhas de Nós na Confiabilidade 1

Seja $f(t)$ a probabilidade de um nó falhar em um intervalo de tempo t.

Seja $t1$ = Intervalo crítico na confiabilidade 1.

Seja $P_1(F)$ a probabilidade de perda definitiva de uma MSG (distribuída por k nós, além do original) na confiabilidade 1.

Temos então: $P_1(F) = f(t1)^k$.

4.9.2 Análise das Falhas de Nós na Confiabilidade 2

Seja $t2$ = Intervalo crítico na confiabilidade 2.

Seja $P_2(F)$ a probabilidade de perda definitiva de uma MSG (distribuída por k nós, além do original) na confiabilidade 2.

Temos então: $P_2(F) = (f(t2))^k$.

4.9.3 Análise Comparativa da Falha dos Nós nas Duas Confiabilidades

Considerando a situação mais provável, com $k = 1$ na confiabilidade 1 e $k = 2$ na confiabilidade 2, temos:

$P_1(F) = f(t1)$ e $P_2(F) = (f(t2))^2$.

Sendo $t1$ e $t2$ relativos a um intervalo entre o envio de uma MSG e o envio de um ACK, pode-se considerar $t1 = t2$ $\big($e, consequentemente, $f(t1) = f(t2)\big)$.

Logo: $P_2(F) = (f(t2))^2 = (f(t1))^2 = (P_1(F))^2$.

Assim, a probabilidade de perda definitiva da mensagem é consideravelmente menor no esquema com confiabilidade 2, como vemos na Figura 4.13.

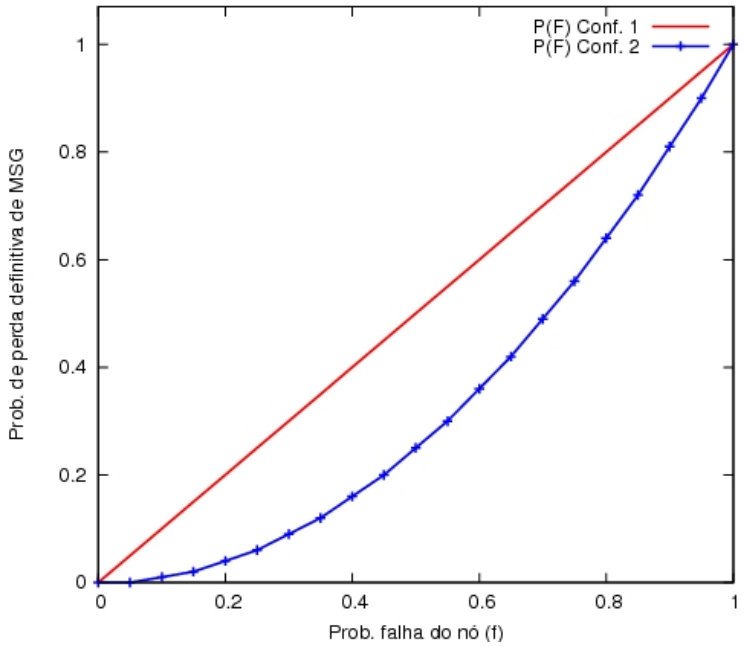

Figura 4.13: Probabilidade de perda definitiva da mensagem nos dois níveis de confiabilidade.

4.10 Resultados Simulados para a Falha dos Nós nos Dois Níveis de Confiabilidade

A Seção 4.9.3 apresenta um resultado simplificado para a análise da falha dos nós. Este resultado resume em si uma relação do comportamento linear do nível 1 de confiabilidade em relação a um comportamento "de segunda ordem"da confiabilidade 2. Isto se dá em função da distribuição da responsabilidade de armazenamento da mensagem no intervalo crítico entre dois nós, em lugar de um. Assim, temos que ter a falha quase simultânea de dois nós, dentro do intervalo crítico, para termos a perda da mensagem. Esta falha simultânea, no modelo probabilístico, se manifesta como uma multiplicação, o que leva a este resultado.

A seguir, temos os resultados das simulações, incluindo as falhas de enlaces. Foram mantidos os mesmos parâmetros das simulações anteriores, $p = 0,63$, $q = 0,79$, $R = 5$, $h = 20$ e 3 execuções de 5×10^7 rodadas cada. A falha dos nós variou

de 0% a 18%, o que significa f variando de 100% a 82%.

Como esperado, pode-se observar pela Figura 4.14 que o nível de confiabilidade 2 suporta muito melhor o aumento da probabilidade de falha de um nó no intervalo crítico. Isto ocorre tanto para o sucesso do protocolo quanto para a taxa de entrega. O valor de f em 90%, por exemplo, mantém a taxa de entrega da confiabilidade 2 em pouco mais de 70%, enquanto a confiabilidade 1 é menor do que 10%.

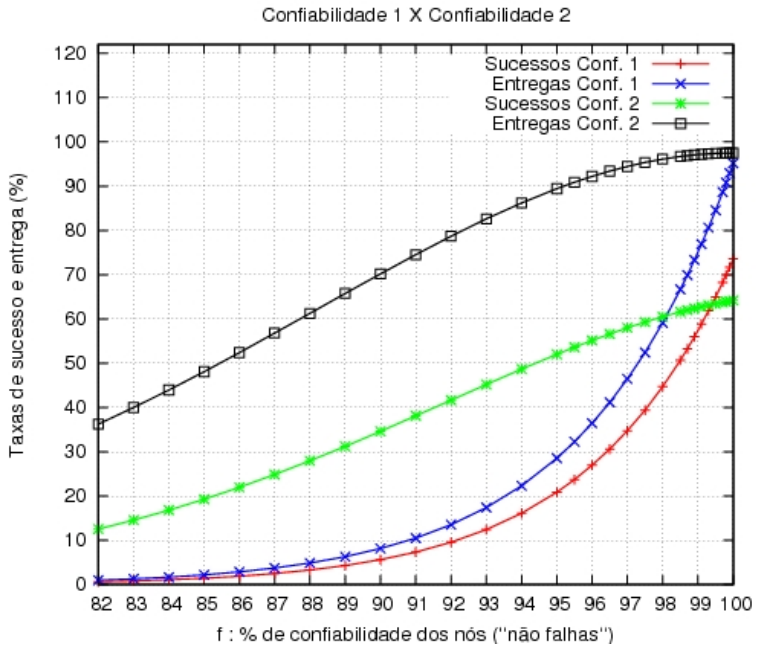

Figura 4.14: Análise da falha dos nós nos dois níveis de confiabilidade.

4.11 Resumo

Este capítulo mostrou, através de modelos matemáticos probabilísticos e de simulações, os ganhos obtidos com os níveis 1 e 2 de confiabilidade propostos neste trabalho.

Foi discutido o modelo de erros de transmissão utilizado no TOSSIM [24] e suas relações com os tamanhos de mensagens utilizados pelo CTCP. Foram calculadas probabilidades típicas para o sucesso no envio de mensagens e ACKs, para os cenários utilizados nas simulações dos próximos capítulos.

De posse dos modelos, do simulador e das probabilidades típicas, foi feita uma análise comparativa entre 3 modelos de transporte: um único ACK fim-a-fim, con-

fiabilidade 1 e confiabilidade 2. Foi observada e quantificada a maior capacidade de entrega de mensagens obtida com o uso do CTCP.

Foi feita uma análise do número ideal de tentativas de retransmissões e valores foram sugeridos para utilização no protocolo.

Por fim, foi feita uma análise do impacto da falha dos nós em intervalos críticos sobre os 2 níveis de confiabilidade. Observou-se a maior robustez do nível 2 de confiabilidade.

Este capítulo foi de grande importância no estudo, refinamento e aprimoramento do protocolo aqui proposto. Seus resultados permitiram as primeiras publicações [19, 18] que validam o trabalho.

Entretanto, uma implementação do protocolo e sua execução em uma plataforma de sensores permitem uma avaliação bem mais real do CTCP. Questões relativas ao nível físico, acesso ao meio, roteamento e consumo de energia são importantes na análise de um novo protocolo.

Diante disto, o CTCP foi implementado em uma plataforma de sensores e testado em redes de sensores simuladas, conforme descrito nos dois capítulos que se seguem.

Capítulo 5

Implementação do CTCP no TinyOS

A implementação do CTCP tem por objetivo avaliar empiricamente os algoritmos propostos. Eles foram implementados em NesC e executados em um dos sistemas operacionais mais utilizados para sensores sem fio, o TinyOS. Descrevemos, nas primeiras seções, uma breve introdução ao TinyOS. A seguir detalhamos o algoritmo que gerou a implementação do protocolo de transporte CTCP e os testes de sanidade que validam a implementação.

5.1 Sistema Operacional TinyOS

O sistema operacional TinyOS foi desenvolvido pela Universidade de Berkeley para os sensores Mica [22]. Este sistema operacional está se tornando um padrão de facto nas RSSF. São exemplos de arquiteturas de nós sensores que utilizam o TinyOS: a família Mica (Mica, Mica2, MicaZ) [74], Telos [75], BTNode [76] e CodeBlue [77]. O TinyOS é um sistema operacional simples e compacto, baseado em eventos, desenvolvido para atender aos requisitos das RSSF. Para diminuir o consumo de memória e processamento, o TinyOS é compilado em conjunto com o software da aplicação em um único binário, que contém somente os módulos necessários ao funcionamento da aplicação. Isto evita que o nó sensor carregue consigo bibliotecas e módulos que não são utilizados. Além disso, a criação de um único executável aumenta o desempenho do nó sensor, pois evita a carga dinâmica de bibliotecas e todos os problemas

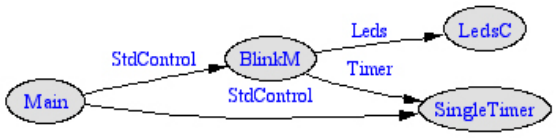

Figura 5.1: Documentação da aplicação Blink.

associados. A linguagem de programação utilizada no TinyOs é um dialeto da linguagem C chamado NesC [78]. No NesC, os programas são compostos de módulos, que se comunicam entre si por interfaces. A Figura 5.1 mostra um exemplo, onde as elipses indicam os módulos e as setas representam as interfaces. No exemplo, o módulo BlinkM utiliza os serviços providos pelo módulo SingleTimer, que fornece abstrações para acesso aos temporizadores do processador. Os módulos dos quais as setas se originam utilizam os serviços fornecidos pela interface que dá nome à seta. Estes serviços são implementados pelos módulos apontados pelas setas. O módulo SingleTimer, por exemplo, implementa a interface Timer, enquanto o módulo BlinkM utiliza a implementação da interface Timer provida pelo módulo SingleTimer.

As interfaces definem comandos e eventos que permitem a comunicação entre módulos. Os módulos que utilizam a interface executam os comandos definidos pela mesma. Os módulos que implementam a interface, por outro lado, notificam a ocorrência de eventos aos módulos que as utilizam. Na Figura 5.1, o módulo BlinkM utiliza comandos da interface Timer para requerer serviços ao módulo SingleTimer, enquanto o módulo SingleTimer notifica o término da execução utilizando os eventos definidos pela interface. As interfaces também são utilizadas no TinyOS para o tratamento de eventos de hardware.

Cada módulo, por sua vez, pode ser constituído por um conjunto de módulos. Isto ocorre com componentes complexos, como as implementações das camadas de roteamento e de controle e acesso ao meio. Esta divisão permite a divisão da implementação de um grande serviço em partes menores. Assim, um serviço pode ser visto como um grande módulo, apesar de ser composto internamente por vários deles. A camada de roteamento, por exemplo, define um módulo chamado MultiHopRouter, que é composto por dois módulos internos, chamados MultiHopEngineM e MultiHopLEPSM.

5.2 Modelo de Programação do TinyOS

Alguns eventos, como aqueles referentes a interrupções de hardware, devem ser tratados imediatamente. Assim, é preciso que os eventos e os comandos sejam executados rapidamente, para que a interrupção seja tratada a tempo. Para suportar cálculos que demandam maior tempo de processamento, o NesC utiliza o conceito de tarefas. As tarefas podem ser interrompidas a qualquer momento por eventos, e retornam a sua execução após o término da execução do evento que lhe interrompeu.

Como os eventos e comandos devem ser executados rapidamente (para atender corretamente aos eventos de *hardware*), o TinyOS emprega um modelo de programação orientado a eventos, onde as funções dos módulos são executados de forma assíncrona. Tomemos como exemplo o envio de mensagens para a rede, que é implementado pelo comando *send* e pelo evento *sendDone*. O primeiro é chamado para sinalizar que um módulo deseja enviar um novo pacote. Para evitar que a execução fique parada até que o pacote seja enviado, o comando *send* retorna imediatamente, sinalizando apenas se o pacote foi colocado na fila para envio ou se a fila está cheia. O fim da transmissão de dados é notificado ao módulo que requisitou a comunicação utilizando o evento *sendDone*. Da mesma forma, qualquer operação no TinyOS que não pode ser realizada instantaneamente é implementada de forma assíncrona, geralmente utilizando uma tarefa. Um comando escalona a execução de uma tarefa, que notifica o fim da sua execução através de um evento. Como vemos a seguir, a implementação do CTCP segue este paradigma.

5.3 Modelo de Comunicação do TinyOS

A pilha de protocolos do TinyOS possui características peculiares que lhe diferenciam dos modelos de referência normalmente utilizados em redes de computadores. Como exemplo, o TinyOS permite que aplicações utilizem apenas os componentes da pilha que são necessários para uma dada aplicação. Caso todos os nós consigam ouvir uns aos outros, por exemplo, pode-se optar por retirar o módulo de roteamento, uma vez que toda a comunicação ocorre em um único salto. Podemos dividir a comunicação no TinyOS em cinco camadas, da superior para a inferior: Aplicação, roteamento, AM, controle e acesso ao meio (MAC) e camada física. Toda comunicação ocorre

em um modelo orientado a pacotes, onde não é garantida a entrega dos dados. A função de cada camada se encontra detalhada a seguir.

- Aplicação: Implementa a lógica da aplicação, requisitando a transmissão de mensagens diretamente à camada de roteamento ou à camada de controle e acesso ao meio. Como no TinyOS, os programas não são obrigados a seguir uma arquitetura rígida de comunicação. A aplicação pode requisitar o envio de pacotes diretamente à camada MAC ou à camada AM. Neste caso, entretanto, a aplicação deve reimplementar alguns serviços das camadas que foram retiradas.

- Roteamento: Constrói as rotas e fornece primitivas para envio de pacotes para a estação base. Por uma decisão de projeto, o módulo de roteamento do TinyOS permite que as aplicações enviem dados somente no sentido da estação base. Caso o nó deseje enviar dados para um dos seus vizinhos, este deve requisitar tal transmissão diretamente para as camadas inferiores. A camada de roteamento ainda provê mecanismos para fusão e agregação de dados durante os saltos da comunicação, bem como uma fila de pacotes a serem transmitidos. Esta é a camada que mais consome memória (devido à fila de pacotes), e por isso é geralmente removida quando a aplicação requer apenas transmissões em um único salto.

- AM (*Active Message*): A camada AM implementa um classificador, que direciona os pacotes para módulos específicos de acordo com um tipo de cabeçalho. Ao contrário dos modelos ISO/OSI e TCP/IP, onde o despacho dos dados para a aplicação correta ocorre acima do nível de roteamento utilizando o conceito de portas, no TinyOS esta tarefa é feita abaixo do roteamento. Em geral, os dados são despachados para o módulo que implementa o roteamento para que este decida o destino do pacote. Entretanto, certos módulos podem optar por gerenciar alguns tipos de pacotes de forma particular.

- Camada MAC: Esta camada implementa um protocolo CSMA/CA, com o uso opcional de mensagens de confirmação (ACK). Esta camada não implementa nenhum tipo de fila de pacotes, pois o roteamento é responsável por tal tarefa. Outra característica marcante do MAC no TinyOS é o emprego de ciclos de

operação, onde o rádio é desligado em intervalos regulares de tempo, tendo em vista a economia de energia. A camada MAC ainda é responsável pelo enquadramento dos dados, detecção de erros e retransmissão de quadros.

- Camada física: A camada física provê um fluxo contínuo de bytes (nas arquiteturas Mica2, MicaZ e Telos) ou de bits (na arquitetura Mica) para a camada MAC, que deve processá-los e discernir dados úteis do ruído do meio. Para tanto, a camada física também provê leituras da potência do sinal recebido. Em geral, a implementação do TinyOS utiliza abstrações que permitem que o código dos protocolos e aplicações sejam independentes do hardware utilizado.

A grande maioria do código escrito para este SO é independente de plataforma. As camadas MAC e física, entretanto, possuem implementações diferentes para cada plataforma suportada pelo TinyOS por serem fortemente relacionadas ao hardware. A camada de roteamento, por outro lado, é em geral, independente de plataforma.

O CTCP é um protocolo de transporte, que se propõe a cobrir a lacuna da camada de transporte existente na arquitetura TinyOS. Da mesma maneira que o protocolo de rede, também é independente de plataforma.

5.4 Roteamento do TinyOS

O TinyOS provê uma arquitetura de roteamento, descrita em [79], que facilita a criação de novos protocolos de roteamento e define interfaces que devem ser implementadas por eles. A arquitetura também simplifica a codificação da aplicação. Para que troquemos o protocolo de roteamento utilizado em uma aplicação é necessário somente substituir o nome do módulo em um arquivo de configuração.

A arquitetura de roteamento é composta de dois módulos. O primeiro, chamado MultiHop-Send, gerencia as filas de pacotes, escalona a transmissão de dados e interage com a aplicação. O Segundo módulo, chamado MultiHopRoute, implementa os protocolos e algoritmos de manutenção da tabela de roteamento e estabelecimento de rotas. Os protocolos de roteamento provêm três interfaces (primitivas) de comunicação com a aplicação, chamadas Send, Receive e Intercept, que são utilizadas para enviar pacotes, receber pacotes e modficar pacotes em trânsito, respectivamente. A interface *Intercept* ainda permite que o módulo que

implementa a aplicação ou outros módulos específicos realizem a agregação, fusão e descarte de dados em cada salto da comunicação. As mensagens de roteamento são do tipo AM_MULTIHOPMSG, e são enviadas pelo AM para o módulo MultiHopRoute. Mensagens de outros tipos são enviadas para o módulo MultiHopSend, que decide se estas são entregues para um módulo no nó atual ou são repassadas para outro nó.

Para rotear os pacotes até a estação base, que no nosso caso é a raiz da árvore de roteamento, utilizamos o MintRoute [22]. O CTCP parte do pressuposto que a rota escolhida pela camada de rede seja bidirecional, o que normalmente acontece. Por isso, acreditamos que nosso protocolo seja capaz de trabalhar sobre vários outros protocolos da camada de rede, tais como CLDP [42], TinyAODV [80], ou BVR [81].

5.5 TOSSIM

O TOSSIM (TinyOS SIMulator) [24, 82] é um simulador de eventos discretos para RSSFs que usam o TinyOS [22]. Ao invés de compilar a aplicação para o mote, usuários podem compila-la para o ambiente do TOSSIM, que a executa em um PC. O TOSSIM permite a depuração, os testes e a análise dos algoritmos em ambientes controlados. Como o TOSSIM é executado em um PC, os programadores podem examinar seus códigos TinyOS através de depuradores e outras ferramentas de desenvolvimento utilizadas em programas para PC. O objetivo primário do TOSSIM é prover uma simulação com alta fidelidade para o TinyOS. Por esta razão, TOSSIM prefere focar na simulação e execução do sistema operacional a simular o mundo real. O código usado durante as simulações pode ser executado diretamente no mote. O TOSSIM permite simular milhares de nós sensores simultaneamente. Na simulação, cada nó sensor executa o mesmo programa TinyOS.

5.6 CTCP no TinyOS

5.6.1 Estrutura do CTCP

O CTCP foi implementado em três módulos diferentes: núcleo, gerência de confiabilidade e gerência de congestionamento. A Figura 5.2 ilustra a interação entre

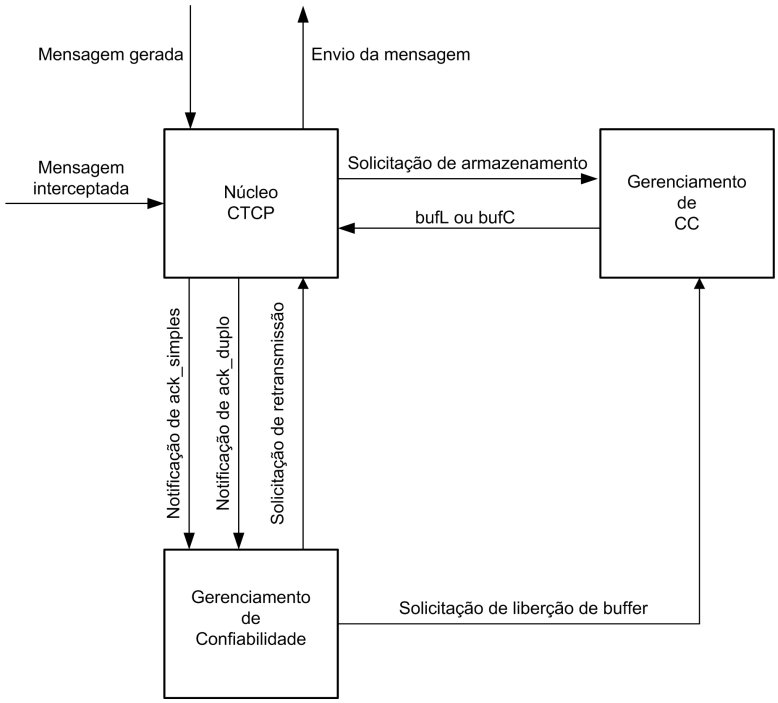

Figura 5.2: Módulos da implementação do CTCP.

eles.

Núcleo do protocolo

O núcleo do protocolo implementa o serviço de recepção e envio de pacotes. Além disso, possui as configurações de parâmetros globais, tais como, o intervalo de controle do temporizador, o tempo de espera pela convergência do roteamento, o número de retransmissões permitidas, o número de nós da simulação, o número de seqüência do primeiro pacote e o tamanho do *buffer* local.

O algoritmo 4 nos mostra a estrutura principal do núcleo do CTCP.

O número de seqüência do primeiro pacote é calculado randomicamente durante a inicialização do protocolo. Logo a seguir, o protocolo entra em estado de espera aguardando chamada da camada de aplicação ou da camada de enlace. Um pacote recebido da camada de aplicação (normalmente leituras) é encapsulado, armazenado e enviado para a camada de rede. O processo de encapsulamento que consiste em preencher cada um dos campos do cabeçalho de transporte foi descrito na Seção 3.3.4. Um pacote recebido através da camada de rede é analisado e redistribuído para outros módulos de acordo com o seu tipo. Assim, cada nó pode receber, da camada de enlace, três tipos de pacotes: pacotes START, ACK (sim-

```
Algoritmo: Núcleo CTCP

se Recebe interrupção de entrada de dados então
    se interrupção oriunda da camada de aplicação então
        monta pacote;
        posição = solicita armazenamento do pacote;
        marca bit de controle (posição)=1;
    senão
        Intercepta mensagem;
        caso pacote = START
            AlteraStatusNoCorrente(FLAG=1);
        caso pacote = ACK_simples_STOP
            AlteraStatusNoCorrente((FLAG=0);
        caso pacote = ACK_simples
            notifica o módulo de gerenciamento de confiabilidade;
        caso pacote = ACK_duplo
            notifica o módulo de gerenciamento de confiabilidade;
        caso pacote = dados
            posição = solicita armazenamento;
            faz alterações necessárias (se houverem);
            marca bit de controle(posição)=1;
        verifica se está liberado para transmitir;
        se FLAG=1 então
            envia os pacotes com (bit de controle=1) para a camada de rede;
    se Recebe notificação de estouro do temporizador então
        retransmite (msgindicada);
```

Algoritmo 4: Estrutura do Núcleo do CTCP.

ples, duplo ou com STOP) e pacotes de dados para reencaminhamento. Por outro lado, os dados oriundos da camada de aplicação geram segmentos de transporte que devem ser entregues à estação base.

Módulo de Gerenciamento do Congestionamento

Este módulo é responsável por controlar o estado de saturação dos *buffers*, além de executar o armazenamento da mensagem. Interage com o núcleo do protocolo informando o estado de saturação. É importante ressaltar, que a área de *buffer* é uma só. O CTCP não trabalha com um *buffer* de entrada e outro de saída. Todas as mensagens são armazenadas na mesma área, o que evita a cópia destas, de um *buffer* para outro. Entretanto, é preciso diferenciar as mensagens em *buffer*, pois existem aquelas que já foram processadas e precisam ser enviadas e aquelas que nem sequer passaram pelo processamento inicial. Assim, foi criado um bit de controle para cada uma das posições do *buffer*. Este bit é igual a 1 para as mensagens que já foram processadas. O algoritmo 5 nos mostra a estrutura principal do gerenciamento de

congestionamento do CTCP.

```
Algoritmo: Gerenciamento do Congestionamento do CTCP

caso Recebe solicitação de armazenamento
    se (ocupação do buffer)¡(patamar T) então
        armazena pacote no buffer;
        envia notificação de buffer livre(bufL);
        retorna (posição de armazenamento);
    senão
        armazena pacote no buffer;
        envia notificação de buffer cheio(bufC);
        retorna (posição de armazenamento);
caso Recebe solicitação de liberação de buffer
    Procura pacote indicado no buffer;
    se achou então
        apaga pacote do buffer;
        marca bit de controle=0;
    se (ocupação do buffer)¡(patamar T) então
        envia notificação de buffer livre(bufL);
    senão
        envia notificação de buffer Cheio(bufC);
```

Algoritmo 5: Estrutura do Gerenciamento do Congestionamento do CTCP.

Módulo de Gerenciamento de Confiabilidade

Este módulo é responsável por garantir que as mensagens geradas ou que passam por este nó chegam íntegras à estação base. Para isso controlam os temporizadores e solicitam retransmissões. Trata os reconhecimentos simples e duplos. O algoritmo 6 descreve a estrutura de gerenciamento de confiabilidade do CTCP.

```
Algoritmo: Gerenciamento de Confiabilidade

caso Recebe notificação de ack_simples
    se confiabilidade = nivel1 então
        para temporizador ack_simples;
        envia solicitação de liberação de buffer;
    se confiabilidade = nivel2 então
        inicia temporizador para esperar pelo ACK_duplo;
caso Recebe notificação de ack_duplo
    para temporizador ack_duplo;
    envia solicitação de liberação de buffer;
caso Estouro Temporizador
    envia solicitação de retransmissão(msgindicada);
```

Algoritmo 6: Estrutura do Gerenciamento de Confiabilidade do CTCP.

5.7 Testes de Sanidade/Fumaça

O teste de sanidade (ou teste de fumaça) é um teste fundamental utilizado para avaliar rapidamente a validade de uma resposta ou de um cálculo. Em matemática, por exemplo, ao multiplicar por três ou nove, deve-se verificar se a soma dos algarismos do resultado é um múltiplo de 3 ou 9.

Em Ciência da Computação, os testes de sanidade representam uma breve análise da funcionalidade de um *software*, sistema, ou cálculo, com a finalidade de garantir que o sistema ou metodologia funcione como esperado. É freqüentemente usado antes de uma rodada de testes mais exaustiva [83].

O teste de sanidade do CTCP foi realizado sobre o *TinyOS Simulator* (TOSSIM) [24]. Criou-se uma rede de cinco nós, alinhados, como mostra a Figura 5.3. Somente o último nó, nó quatro, gera mensagens de 10 em 10s. Nenhuma taxa de erro foi inserida nos canais da rede, de maneira que todos os pacotes transmitidos chega, sem erros, ao seu destino. O raio de transmissão de cada nó foi mantido em seu valor *default*, 50 pés (15,24 m).

Durante o teste de sanidade, o simulador trabalhou com protocolos *default*, tanto na camada de rede quanto na do enlace. O valor utilizado para o intervalo de controle do temporizador foi arbitrário e o número de retransmissões permitidas foi igual a 4.

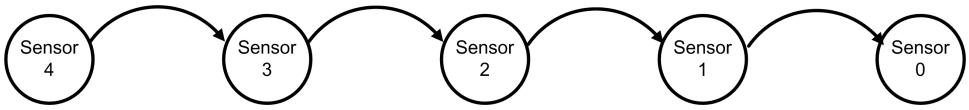

Figura 5.3: Topologia utilizada para realizar os testes de sanidade.

Após as primeiras simulações constatou-se a necessidade de introduzir um tempo inicial, da ordem de 150 segundos, para que o algoritmo de roteamento pudesse convergir. Antes de introduzir este tempo inicial as mensagens geradas durante esta fase eram perdidas. Notou-se, claramente, a necessidade de ajuste do intervalo de controle do temporizador. Este ajuste foi desenvolvido e pode ser encontrado na Seção 3.3.2.

5.8 Resumo

Este capítulo apresentou a implementação do CTCP para o sistema operacional TinyOS. A implementação do CTCP neste sistema operacional, que é quase um padrão de facto em RSSF, permite que o protocolo proposto seja executado nas plataformas mais utilizadas em RSSF. Foi apresentada a arquitetura de módulos do CTCP e as decisões de implementação. A implementação do CTCP procurou, sempre que possível, minimizar a quantidade de código necessária e manter inalteradas as interfaces e nomes de módulos que o CTCP exporta. Os testes de sanidade apontaram deficiências no intervalo de controle do temporizador e no tempo de convergência do algoritmo de roteamento. Estas deficiências foram sanadas antes de iniciar a fase de validação que pode ser encontrada no Capítulo 6.

Capítulo 6

Análise dos Resultados

No Capítulo 4 e em [19], o mecanismo de confiabilidade foi analisado probabilisticamente. Neste capítulo, faz-se uma análise da implementação do CTCP sobre o TinyOS *Simulator*. Estes resultados estendem aqueles apresentados no trabalho [84]. As métricas utilizadas foram entrega de pacotes à estação base, latência e consumo de energia. Em seguida, foi avaliado o algoritmo de detecção e prevenção de congestionamento. Por fim, com base nos resultados obtidos foram feitas comparações entre o CTCP e outros protocolos de transporte existentes.

6.1 Ambiente das Simulações e Parâmetros Utilizados

Foram utilizadas, nas simulações, redes com 25, 49 e 100 nós sensores, distribuídos em áreas de $40 X 40 pés$, $60 X 60 pés$ e $90 X 90 pés$, respectivamente. Um espaçamento de 10 pés $(3,05m)$ foi mantido entre os nós em todas as topologias. Estas topologias estão representadas em conjunto na Figura 6.1. Os círculos pretos representam os nós fonte em cada uma delas e o círculo cinza o nó sorvedouro.

O alcance do rádio de comunicação dos sensores também foi mantido constante. Desta maneira, o aumento do número de nós aumenta também o número de saltos até a estação base.

Durante as simulações, um único sensor faz leituras e gera mensagens, trata-se do nó fonte. O nó fonte se situa em um dos vértices da área ocupada pelos nós. Ele envia, ao nó sorvedouro, um pacote de leitura a cada 50 segundos do simulador. O

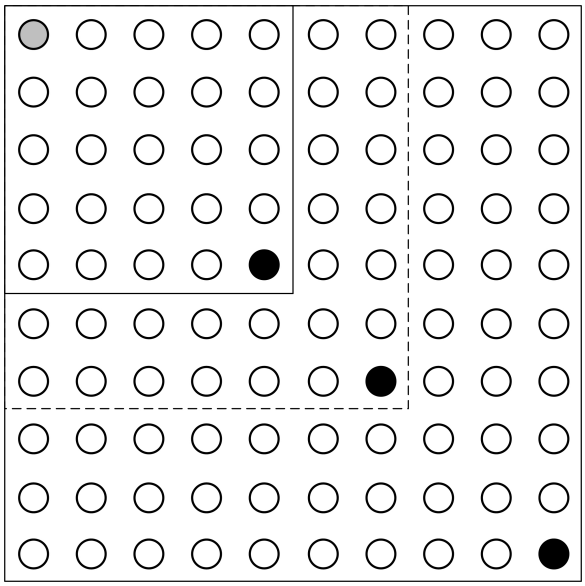

Figura 6.1: Topologias de 25, 49 e 100 nós.

nó sorvedouro se situa no vértice oposto (Figura 6.1).

O raio de transmissão de cada nó é de 50 pés $(15,24m)$. Combinado com a taxa de erros, isto significa que cada nó transmite seu sinal em uma área circular com 50 pés de raio, com a taxa de erro aumentando a medida que se afasta do centro. Como visto na Seção 3.3.4, o pacote de dados tem 208 bits de tamanho enquanto o pacote de reconhecimento possui 64 bits.

Para rotear os pacotes até a estação base, foi utilizado o protocolo de roteamento padrão do TOSSIM, chamado MintRoute. O MintRoute é um protocolo de roteamento proativo no qual os nós enviam mensagens de roteamento periódicas para informar sobre o seu estado local. Observou-se, nas simulações, que as rotas utilizadas do nó fonte ao sorvedouro ficam próximas à diagonal que une estes dois nós (ou vértices) e, em algumas situações, salta um nó.

O TinyOS, e consequentemente o TOSSIM, usam o CSMA como protocolo padrão da camada de enlace.

As simulações foram executadas durante 1000 segundos do simulador.

Os intervalos de controle do temporizador foram ajustados conforme especificações desenvolvidas na Seção 3.3.2 e o número máximo de retransmissões foi fixado em 10, conforme resultado das Seções 4.7.1 e 4.8.1.

As maiores funções de um protocolo de transporte para redes de sensores são

controle de congestionamento, garantia de entrega e conservação de energia que pode ser alcançada através de eficientes controles de congestionamento e garantia de entrega [47].

Para avaliar o desempenho do CTCP, usou-se as seguintes métricas:

- Fração de pacotes entregues com sucesso;
- Consumo de energia;
- Latência.

6.2 Fração de Pacotes Entregues com Sucesso

Para avaliar o rendimento de uma aplicação executada sem um serviço de transporte confiável em comparação com a mesma aplicação utilizando os níveis 1 e 2 de confiabilidade, foram executadas simulações segundo os parâmetros descritos na Seção 6.1.

A confiabilidade é medida como a fração de pacotes transmitidos e efetivamente recebidos na presença de erros de bits e falha de nós.

A Figura 6.2 ilustra a taxa de entrega em três situações diferentes. Na primeira, a aplicação roda diretamente sobre a camada de rede, sem protocolo de transporte. Na segunda situação, o CTCP foi introduzido e configurado para confiabilidade nível 1. E nas últimas séries de simulações, o CTCP foi ajustado para nível 2 de confiabilidade. Para cada nível de confiabilidade, as simulações foram executadas para as três topologias: 25, 49 e 100 nós.

Na rede com 25 nós, a entrega é de $28,88\%$ sem nenhum protocolo de transporte. A mesma aplicação, na mesma rede, executando o CTCP com confiabilidade nível 1, entrega 70% dos pacotes, enquanto o nível 2 chega a 94%. Note que nestas simulações, o protocolo de transporte é imprescindível, pois mesmo para uma aplicação que possui grande redundância de dados, uma entrega de apenas $28,88\%$ dos pacotes pode ser inviável. A utilização do nível 2 de confiabilidade é apropriado para aplicações onde existe pouca redundância de pacotes.

Nas redes com 49 e 100 nós, podemos notar que existe uma queda na taxa de entrega de pacotes. Esta queda se dá principalmente em função da latência da rede

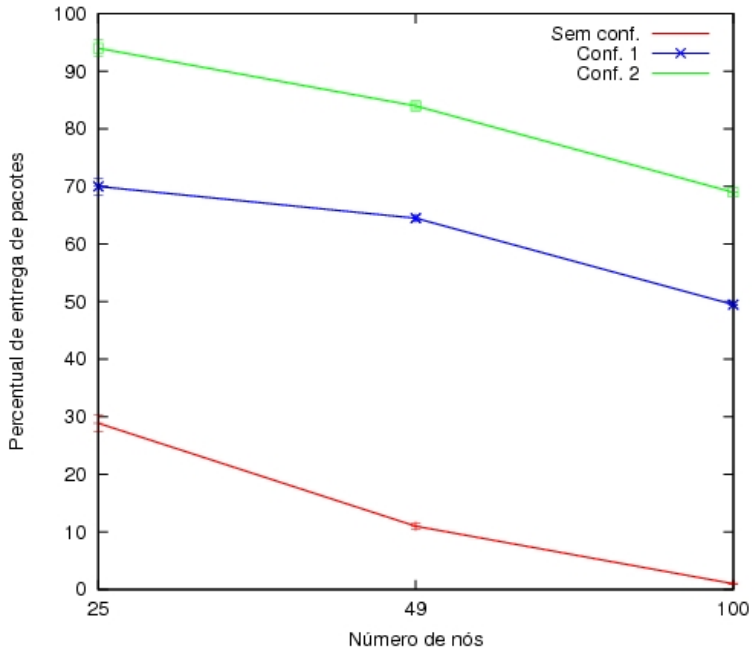

Figura 6.2: Probabilidade de entrega de pacotes.

que é analisada na Seção 6.4.

Os arquivos de *logs* das simulações mostram que o CTCP nível 2 pode alcançar 100% de entrega de pacotes com alguns ajustes na configuração do temporizador. Neste caso, o total de pacotes não entregues está relacionado a uma alta latência e não à perda permanente.

É importante ressaltar que as simulações foram executados com apenas um nó origem fazendo leituras e, consequentemente gerando pacotes, a cada 50 segundos do simulador. Se aumentarmos o número de nós que fazem leituras e geram pacotes, passamos a ter uma grande probabilidade de congestionamento. Neste caso, é muito importante o uso do mecanismo de controle de congestionamento descrito na Seção 3.3.3. O controle de congestionamento reduz o número descartes e retransmissões devidos a *buffer* cheio, conseqüentemente reduzindo a latência na rede.

6.3 Consumo de Energia

Na Seção 6.2, podemos ver que o CTCP é capaz de prover altos níveis de confiabilidade. Como a energia é um recurso escasso nas redes de sensores, é necessário medir o aumento do consumo de energia gerado pela inserção do protocolo de transporte.

Para as simulações descritas anteriormente (sem confiabilidade, nível 1 e nível 2), foram contados o número de total de pacotes transmitidos. Considerou-se duas classes distintas de pacotes: pacotes de dados e pacotes de reconhecimento (quando existiam). Como já mencionado, o tamanho do pacote de dados é de 208 bits incluindo a mensagem da camada de aplicação (a leitura do sensor), o cabeçalho do CTCP, camada de rede e enlace. O pacote de reconhecimento, que é enviado diretamente da camada de transporte para a camada de enlace (não passa pela camada de rede) é destinado ao endereço de enlace (MAC *address*) e tem apenas 64 bits.

Assume-se que o rádio dissipa $50\,nJ/bit$ (*nanojoule* por bit) para acionar o circuito de transmissão ou recepção e $0,1\,nJ/(bit \times m^2)$ para amplificar a transmissão de maneira a alcançar um SNR [85] aceitável.

A área em questão (Figura 6.3) é dada por:

$A = \pi \times (50 pés)^2 = \pi \times (15,24m)^2 = 729,66\,m^2$

O consumo K do circuito é:

$K = (50 \times b)\,nJ$

Considere C_X como o consumo de X para transmitir b bits, por $A\,m^2$, com $K\,nJ$ de consumo do circuito. Assim, temos:

$$C_X = (0,1 \times A \times b + K) \left[\frac{nJ}{bit \times m^2} \times m^2 \times bit + \frac{nJ}{bit} \times bit \right] \tag{6.1}$$

Os valores resultantes para o consumo de energia por mensagem e ACKs é dada por:

$C_X(MSG) = 0,1 \times 729,66 \times 208 + 50 \times 208 = 25.576,928\,nJ$

$C_X(ACK) = 7.869,824\,nJ$

Baseado nestes valores, calculamos o consumo de energia da camada de aplicação e do CTCP para cada nó. A soma dos consumos individuais é o consumo da rede. A Figura 6.4 mostra o consumo de energia das camadas de aplicação e transporte das três redes diferentes. Todos os valores estão em (mJ).

Como era esperado, a Figura 6.4 mostra um consumo maior de energia para prover um maior índice de confiabilidade. Depois de 1000 segundos, o total de energia gasto na rede é de $12,07\,mJ$ no nível 2, de $6,29\,mJ$ no nível 1 e $2,19\,mJ$ sem protocolo de transporte.

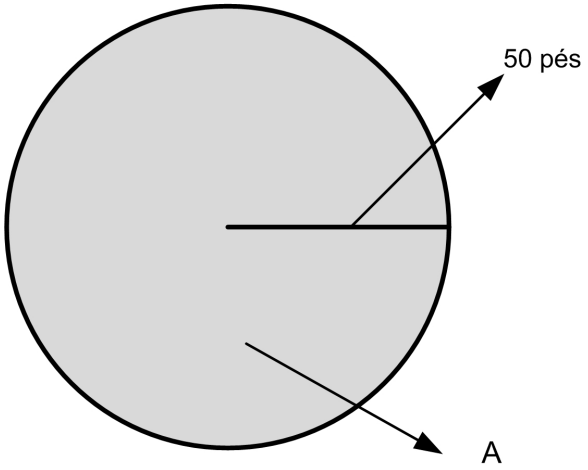

Figura 6.3: Área de alcance de um sensor X.

Estes baixos valores (na ordem de milijoules) ocorrem porque existe somente um nó gerando pacotes na rede e somente 15 mensagens são geradas em cada simulação. Isto é aceitável, uma vez que o objetivo principal da simulação é comparar os três níveis de confiabilidade.

A energia gasta para prover a confiabilidade nível 2 é $1,91$ vezes maior que a energia gasta para prover o nível 1 e $5,51$ vezes o total gasto na simulação sem o protocolo de transporte.

A energia consumida pelas camadas inferiores (rede, enlace e física) é o mesmo em todos os casos estudados. Assim, a sobrecarga proporcional imposta pelo CTCP se torna menos significativa.

6.4 Latência

Considerou-se que a latência de um determinado pacote é a diferença entre o instante de chegada da mensagem na estação base e o instante no qual a mensagem foi gerada. A média aritmética simples destes valores gerou o que chamamos de latência da rede. Deste modo, a Figura 6.5 ilustra a latência de três redes diferentes: com 25, 49 e 100 nós respectivamente. Pode-se concluir, a partir desta figura, que a latência da rede aumenta à medida que aumentamos o número de nós. Tal fenômeno ocorre em função do aumento do número de saltos, ou seja, quanto maior o caminho a ser percorrido mais tempo leva-se para completá-lo.

O valor do temporizador é diretamente proporcional à latência da rede, ou seja,

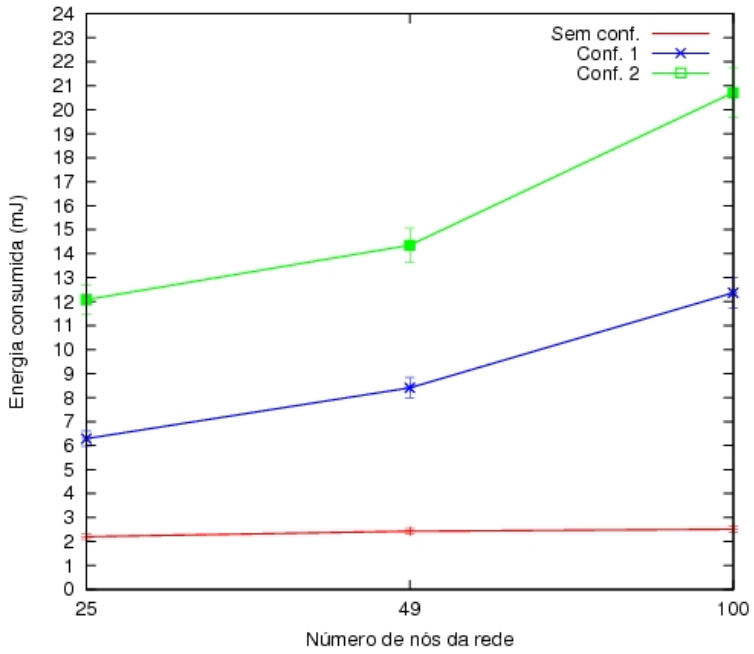

Figura 6.4: Consumo de energia: 25, 49 e 100 nós; sem transporte, níveis 1 e 2.

quanto menor o intervalo de controle do temporizador, menor a latência. Entretanto, como visto na Seção 3.3.2, existe um compromisso entre os valores de temporização utilizados e as retransmissões desnecessárias. Como está apresentado a seguir, o uso dos ajustes deduzidos na Seção 3.3.2, produzem bons resultados.

Apesar de aumentar a latência da rede, um número razoável de retransmissões é fundamental para a obtenção dos níveis de confiabilidade desejados (ver Seções 4.7.1 e 4.8.1). Nas simulações, o uso de $R = 10$, atingiu bons níveis de entrega e, como mostrado a seguir, os valores da latência se mantiveram em bons patamares.

Como era esperado, a latência da rede sem o protocolo de transporte é muito baixa, na ordem de milisegundos. Deve-se considerar que esta latência foi calculada levando-se em consideração somente as poucas mensagens que chegaram.

A latência aumenta à medida que aumentamos o nível de confiabilidade do protocolo. O CTCP nível 2 entrega mais mensagens que o CTCP nível 1, esta diferença na entrega de mensagens acontece principalmente em função de uma maior distribuição de armazenamento das mensagens. O armazenamento distribuído aumenta a possibilidade de uma retransmissão também distribuída.

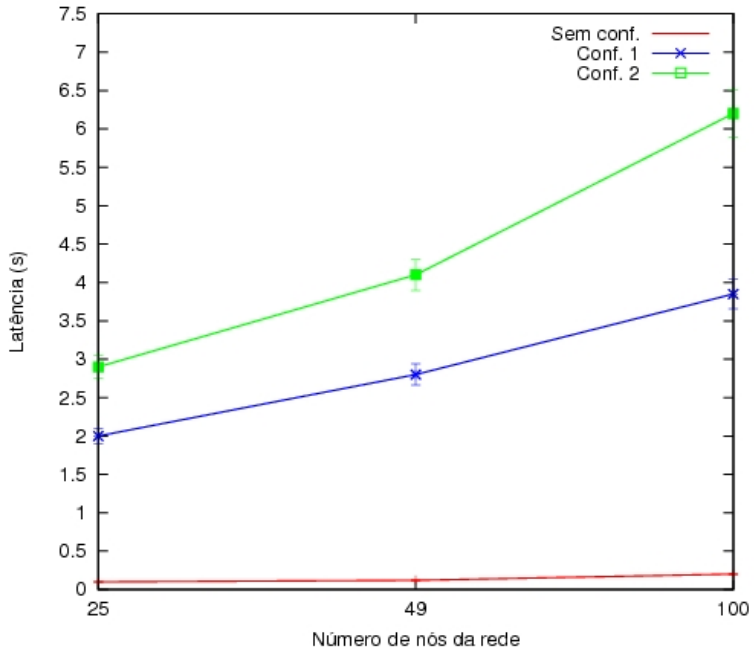

Figura 6.5: Latência: 25, 49 e 100 nós; sem transporte, níveis 1 e 2.

6.5 Controle de Congestionamento

Nesta seção são analisados os conceitos básicos e operacionais do algoritmo de detecção e controle de congestionamento.

Gerou-se um ambiente de simulação com 25 nós (Figura 6.6). A rede foi "sobrecarregada" com um aumento do número de mensagens, de maneira que o nó origem (24) passou a gerar duas mensagens (ou leituras) por segundo. Este nó está situado na extremidade oposta à estação base (nó 0) e é o único gerador de mensagens. A rede executa o nível 1 de confiabilidade do CTCP e este nível não foi alterado durante o tempo de simulação, que foi de 1000 segundos.

Os *buffers* do nível de transporte foram configurados para armazenar até 10 mensagens. O patamar máximo que dispara o envio de mensagens STOP foi configurado empiricamente em 6 mensagens e o mínimo, que gera o envio dos STARTs, em 4. Na Seção 3.3.3 a especificação prevê um único patamar T, mas a implementação mostrou que este amortecimento de 2 mensagens reduziu o número de mensagens STOP/START, sem prejuízo do funcionamento do controle de congestionamento. Observa-se ainda que um reduzido tamanho de *buffers* (10 mensagens) foi suficiente em todas as simulações.

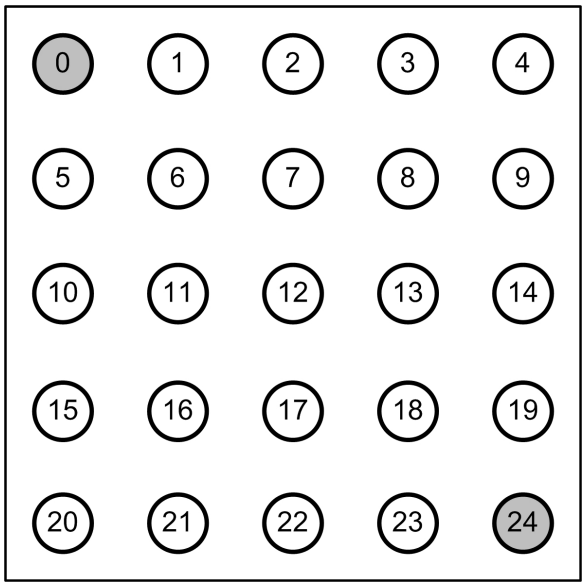

Figura 6.6: Topologia de 25 nós utilizada nas simulações de controle de congestionamento.

Neste primeiro conjunto de execuções da simulação espera-se que a taxa de entrega de mensagens à estação base seja menor em função do alto grau de congestionamento gerado. O arquivo de erro de comunicação dos enlaces utilizado foi o mesmo das seções anteriores.

O principal propósito destas simulações foi medir o número total de pacotes descartados pela rede. Entenda-se por descarte o número de mensagens perdidas por esgotamento de *buffer*. Não são consideradas, neste momento, as perdas relativas a erro de transmissão.

Para efeito de comparação, foram executadas simulações sem controle de congestionamento. Notou-se que os nós intermediários da rede "sobrecarregada" descartam, em média, 46% das mensagens transmitidas.

O algoritmo de detecção e controle de congestionamento descrito na Seção 3.3.3 foi implementado e incluído no código do protocolo e as simulações foram executadas novamente. Não houve nenhuma perda de pacote nos nós intermediários. Isto significa que a detecção e o controle de congestionamento atuou de maneira esperada e eliminou o congestionamento da rede.

O custo observado foi que a "onda" de mensagens STOP chega ao nó fonte, fazendo com que este reduza a taxa de geração de leituras até que a rede se estabilize,

ou seja, até que os nós intermediários reduzam seus *buffers* para ocupações abaixo do patamar e enviem mensagens START no sentido contrário ao fluxo.

Nota-se que o protocolo, por concepção de projeto, privilegia a confiabilidade e garantia de entrega, em detrimento da latência e mesmo da taxa de envio de mensagens por parte do nó fonte.

Para que não ocorram perdas no nó origem ou a referida contenção, é necessário que a rede esteja bem dimensionada à taxa de geração de mensagens. Outra possibilidade é que o nó fonte tenha uma grande capacidade de armazenamento das leituras e medições efetuadas, para absorver as eventuais reduções de taxa impostas pela rede. O que ficou claro nas simulações foi que, uma vez enviadas pelo nó origem, as mensagens nunca foram perdidas por estouro de *buffers* nos nós intermediários.

Todas estas conclusões podem ser extrapoladas para a existência de mais de um nó fonte.

Foi verificado, na prática, a capacidade do CTCP de diferenciar a perda relativa a congestionamento da perda relativa a erro de transmissão, atuando preventivamente contra a ocorrência do primeiro tipo de perda (Seção 3.2).

6.6 Comparação de Resultados

Esta seção tem o objetivo de comparar os resultados do CTCP com os de alguns protocolos de transporte para redes de sensores. A comparação deve ser parcial em função do objetivo de cada protocolo. O CTCP visa resolver dois problemas: entrega garantida e congestionamento. Além disso, o projeto do CTCP não coloca em primeiro plano a otimização da latência de entrega de pacotes, ou seja, no CTCP, cada mensagem gerada pela rede tem a garantia de entrega, independente do tempo necessário para fazê-lo.

Ao analisar o desempenho do protocolo Flush [86], os autores utilizaram a vazão como métrica principal. Considerou-se como vazão o número total de bits recebidos na estação base na unidade de tempo. É importante ressaltar que o protocolo Flush não garante a entrega individual de cada mensagem. Para garantir um certo grau de confiabilidade, o protocolo replica cada uma das mensagens várias vezes, para que pelo menos uma delas chegue à estação base. O protocolo Flush não se preocupa

com o descarte de mensagens, uma vez que existe uma grande redundância destas na rede e seu objetivo principal é gerar uma alta vazão, para que a estação base tenha conhecimento de um determinado evento o mais rápido possível. O CTCP não se baseia na redundância de mensagens. Em contrapartida, as aplicações para as quais o CTCP foi projetado não têm requisitos de entrega imediata, mas sim de garantia individual de entrega das mensagens.

O protocolo PSFQ [55] provê entrega confiável do sorvedouro aos nós, focando a reprogramação dos nós. Entretanto, baseia-se em inundação e redundância, o que potencializa o consumo de energia. O CTCP, por sua vez, utiliza comunicação em *unicast* e não se baseia em múltiplas réplicas de uma mesma mensagem, sendo naturalmente mais econômico. O CTCP pode até mesmo ser aplicado a uma tarefa de reprogramação confiável de nós da rede, sendo, entretanto, adequado à reprogramação de alguns nós específicos, já que atua com conexões fim-a-fim. Com mínimas modificações na abertura de conexões, o CTCP pode prover exatamente os mesmos serviços de transporte, quer esteja atuando no sentido fonte-sorvedouro, quer no sentido contrário.

Para os autores do protcolo STCP [51], o objetivo da camada de transporte é reduzir a latência de cada um dos pacotes. Para medir a latência, foram introduzidas perdas na rede através de descarte de pacotes nos nós intermediários. No STCP, uma pacote leva aproximadamente 2,6s para atravessar uma rede com 100 nós enquanto no CTCP, um pacote que faz o mesmo trajeto, leva 3,8s (nível 1 de confiabilidade). Entretanto, o consumo de energia de toda a rede no STCP é de 56 mJ, enquanto no CTCP é de apenas 11,2 mJ no nível 1 de confiabilidade e 20,8 mJ no nível 2. É possível concluir que estes dois protocolos trabalharam de maneiras diferentes com o compromisso latência X consumo de energia. O STCP foca uma entrega mais rápida por um maior custo energético, enquanto o CTCP preocupa-se em diminuir o consumo de energia na rede. O STCP não implementa o controle de congestionamento.

Outra diferença de projeto existente entre o CTCP e o STCP está na entidade que controla os parâmetros de confiabilidade da conexão. Enquanto o STCP deixa esta configuração inicial a cargo do nó sensor que inicia a conexão, o CTCP prevê que a estação base o fará. O modelo do CTCP é mais adequado, já que o usuário que

interage com a estação base possui o conhecimento global da aplicação em uso e de seus objetivos. Pode-se ter até uma reconfiguração seletiva do nível de confiabilidade de algumas das conexões em curso.

O protcolo CODA [12] foca, exclusivamente, o controle de congestionamento, ignorando a entrega garantida de mensagens individuais. A detecção de congestionamento do CODA é feita através de monitoramento da ocupação do canal. Ao adequar este mecanismo às RSSFs, sobretudo com relação ao consumo de energia, CODA permite a existência de momentos onde muitos pacotes são descartados até que o controle de congestionamento atue. A proposta do CODA não se adequa a longos períodos de congestionamento, limitação não observada nas simulações do CTCP. Por outro lado, CODA obtém altos índices de ocupação do canal e altas taxas de redução do consumo de energia, se aproveitando principalmente dos momentos de carga baixa ou nula na rede e atuando no controle de congestionamento. O CTCP mantém um monitoramento constante de *buffer* e sinalização imediata do congestionamento, evitando, por completo, o descarte de mensagens, agindo preventivamente e não reativamente ao congestionamento. O CTCP obtem índices menores de ocupação do canal e de conservação de energia. Para os momentos de carga leve ou nula na rede, o mecanismo do CTCP se comporta bem, já que mensagens de START e STOP não são utilizadas.

6.7 Resumo

Este capítulo analisou o desempenho do protocolo CTCP implementado sobre o sistema operacional TinyOS. Apesar da utilização do simulador TOSSIM esta análise possui grande relevância, uma vez que o código utilizado na simulação é o mesmo utilizado em sensores reais. A análise de desempenho do mecanismo de confiabilidade do CTCP foi realizada com base em três métricas principais: entrega de pacotes, latência e consumo de energia. Os resultados são satisfatórios e demonstram coerência com os resultados do Capítulo 4. O mecanismo de controle e detecção de congestionamento foi avaliado em função do número de pacotes descartados por esgotamento de *buffer*. Os resultados demonstraram que o CTCP é capaz de detectar a iminência de congestionamento e reagir a este futuro congestionamento evitando os

descartes. Os resultados finais obtidos através da simulação indicam que o protocolo poderá ser utilizado em ambientes reais sem modificações. A comparação do CTCP com outros protocolos de transporte demonstra a existência de uma grande variação de enfoques que reflete a diversidade das aplicações. O projeto de CTCP buscou uma maior independência das camadas inferiores da pilha de protocolos justamente com o objetivo de se adaptar à maioria das classes de aplicações existentes.

Capítulo 7

Conclusões e trabalhos futuros

Esta tese propôs um Protocolo de Transporte Colaborativo para Rede de Sensores sem fio (CTCP) [87] [88]. As características e funcionalidades de tal protocolo permitem-no atender a uma classe de aplicações que necessita de entrega confiável, fornecendo garantia de entrega, bem como alta eficiência em termos de consumo de energia.

Este capítulo apresenta as principais contribuições do protocolo proposto, bem como possíveis direções futuras de pesquisa a serem seguidas. Na Seção 7.1 são descritas as inovações trazidas pela proposta e são ressaltados seus principais benefícios. Na Seção 7.2 são apresentados os trabalhos futuros.

7.1 Principais Contribuições

Esta tese abordou dois dos principais desafios atuais das redes de sensores sem fio que são a entrega confiável de mensagens e o controle de congestionamento. Do ponto de vista de projeto, utilizou-se da especificação formal para descrever o protocolo proposto. Esta técnica validou os algoritmos propostos e facilitou a implementação do protocolo. Do ponto de vista de um protocolo de transporte, a proposta procura incorporar todas as características e funcionalidades necessárias, tanto para as aplicações clientes, como para a infraestrutura de rede, livrando-a da tarefa de aumentar a redundância de dados para que estes cheguem íntegros à estação base. As principais contribuições são detalhadas a seguir.

Garantia de entrega dos segmentos à camada de aplicação da estação

base. Cada um dos segmentos gerados pela rede é unívoco e imprescindível à estação base. Para promover a entrega confiável de segmentos entre um nó e sua respectiva estação base, foi proposto um Algoritmo Distribuído de Confiabilidade Dinâmica que se baseia na distribuição controlada de um mesmo segmento pela rede. Este algoritmo trabalha com um compromisso entre confiabilidade e consumo de energia. Quanto mais alta a confiabilidade requerida maior é o consumo de energia. Ao escolher o nível 1 de confiabilidade do protocolo CTCP obtém-se altos índices de entrega. Entretanto, estes índices não são de 100%, em função da busca por economia energia e da falha de nós sensores. O nível 2 provê praticamente 100% (Seções 4.8 e 6.2) de taxa de entrega, sendo bem mais robusto em presença de falhas distribuídas dos nós (Seções 4.9 e 4.10). Existe, contudo, um maior consumo de energia.

Tem-se ainda a possibilidade de escolha do nível de confiabilidade a ser usado e mesmo de alteração deste nível em tempo de execução. Deste modo, pode-se adequar o funcionamento da rede às necessidades de cada aplicação e mesmo ao tempo de vida útil desejado para a rede.

Dentre as contribuições do CTCP, observa-se que o protocolo oferece entrega confiável, sem gerar um alto grau de redundância de mensagens na rede (como é o caso dos protocolos PSFQ [55] e Flush [86], por exemplo). Além disso, a possibilidade de ajuste dinâmico do nível de confiabilidade, sem necessidade de fechamento e reabertura da conexão, é inovador na literatura relacionada às RSSF.

Habilidade de controle de congestionamento. O CTCP é capaz de detectar e controlar o congestionamento através da diferenciação entre as perdas relativas a erro de transmissão e aquelas geradas por esgotamento de *buffers*. Atuando preventivamente a um eventual esgotamento destes *buffers*, o protocolo elimina os descartes nos nós intermediários da rede. Restam, então, apenas as perdas por erros de transmissão. Para estas perdas, o protocolo oferece recuperação em dois níveis de confiabilidade.

O TCP, por exemplo, apresenta problemas em redes sem fio, uma vez que sua reação é a mesma diante dos dois casos de perda. Assim, conclui-se que esta habilidade de distinção do CTCP é uma importante contribuição da presente proposta.

O CTCP prevê interrupção/liberação explícita, com o uso do próprio ACK para

interromper (STOP) e apenas uma comunicação de um salto para recomeçar a transmissão (START). Este comportamento simples e imediato se mostrou eficiente nas simulações analisadas no Capítulo 6.

Ser independente das camadas subjacentes. Neste trabalho, em nenhum momento se fez qualquer restrição aos protocolos de enlace, acesso ao meio ou roteamento a serem usados. Não há ainda requisitos mínimos que estes devam oferecer. Existe apenas a necessidade da existência de um protocolo de roteamento em execução.

Distribuir funções de confiabilidade e controle de congestionamento, enquanto centraliza parâmetros da conexão. A estação base assume as decisões que dependem do conhecimento dos requisitos da aplicação (confiabilidade requerida) e controla parâmetros com características centrais (ID da conexão). Por outro lado, as funções de controle de congestionamento, implementação da confiabilidade e abertura de conexões estão distribuídas na RSSF.

Proposta de um modelo probabilístico e de um simulador Foi analisado probabilisticamente o mecanismo de confiabilidade proposto e concluiu-se que a recuperação distribuída de erros (salto a salto) aumenta significativamente a probabilidade de entrega de mensagens fim-a-fim. Observou-se ainda que, dividindo a responsabilidade de armazenamento temporário da mensagem entre dois nós adjacentes (nível 2 de confiabilidade), temos uma queda importante da probabilidade de perda definitiva de uma mensagem. Além disso, o armazenamento distribuído de mensagens até que um ACK confirme que a mensagem já se encontra sob responsabilidade do próximo (ou dos 2 próximos) nó(s) demonstra a robustez do protocolo diante de períodos de desconexão.

Os modelos e simulações permitiram sugerir valores apropriados para o número de tentativas de retransmissões de mensagens.

Os modelos probabilísticos e o simulador desenvolvidos se constituíram em importantes ferramentas de análise e desenvolvimento na fase pré-implementação do protocolo. O simulador, na medida em que modela cada evento com uma probabilidade parametrizada e independente e os encadeia dentro da máquina de estados do protocolo, pode até mesmo ser extrapolado e utilizado de forma genérica.

Calibragem do intervalo de controle do temporizador. Este trabalho

propôs e avaliou uma técnica de cálculo do intervalo de controle do temporizador. Com base em técnicas bem conhecidas e testadas, importantes conclusões e adequações foram sugeridas para as RSSF.

7.2 Trabalhos Futuros

Em trabalhos futuros pretende-se:

1. Investigar o efeito do aumento do número de saltos na geração de ACKs múltiplos no modelo de nível 2 de confiabilidade. Serão utilizadas como métricas o aumento do custo de transmissão e de consumo de *buffers* versus aumento na probabilidade de entrega;

2. Analisar a multiplexação de conexões de transporte;

3. Quantificar o tamanho dos *buffers*, bem como do patamar de ocupação para geração de mensagens STOP e START;

4. Especificar as modificações para operação bidirecional do protocolo. Como mencionado na Seção 6.6, mínimas modificações na abertura da conexão permitirão este uso nos dois sentidos (fonte-sorvedouro e vice-versa). O protocolo, após a abertura da conexão, já atende, na íntegra, a este modo de operação;

Referências Bibliográficas

[1] DELICATO, F. C., *Middleware Baseado em Serviços para Redes de Sensores Sem Fio*, Tese de Doutorado, GTA, COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, Brasil, Junho 2005.

[2] TILAK, S., ABU-GHAZALEH, N. B., HEINZELMAN, W., "A taxonomy of wireless micro-sensor network models", *ACM SIGMOBILE Mobile Computing and Communications Review*, v. 6, n. 2, pp. 28–36, Abril 2002.

[3] DE OLIVEIRA CUNHA, D., *Rede sem Fio De Múltiplos Saltos: Protocolos Específicos para Aplicações e Roteamento com Suporte à Diversidade Cooperativa*, Tese de Doutorado, GTA, COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, Brasil, Julho 2008.

[4] CORSON, S., MACKER, J., "Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations", *Request for Comments: 2501*, Janeiro 1999.

[5] HEINZELMAN, W., CHANDRAKASAN, A., BALAKRISHNAN, H., "Energy-Efficient Communication Protocol for Wireless Microsensor Networks". In: *In Proccedings of the Hawaii Conference on System Sciences*, p. 8020, Janeiro 2000.

[6] XU, Y., HEIDEMANN, J., ESTRIN, D., *Adaptive Energy-Conserving Routing for Multihop Ad Hoc Networks*, Relatório Técnico 527, USC/Information Sciences Institute, Outubro 2000.

[7] HAVINGA, P. J. M., SMIT, G. J. M., BOS, M., "Energy-efficient Adaptive Wireless Network Design". In: *The Fifth Symposium on Computers and Communications (ISCC'00)*, p. 502, Antibes, France, Julho 2000.

[8] FEENEY, L. M., NILSSON, M., "Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment". In: *IEEE INFOCOM*, v. 3, pp. 1548–1557, Anchorage, AK, USA, Março 2001.

[9] SHNAYDER, V., HEMPSTEAD, M., RONG CHEN, B., et al., "Simulating the power consumption of large-scale sensor network applications". In: *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems, SenSys*, pp. 188–200, ACM Press, 2004.

[10] FOK, C.-L., ROMAN, G.-C., LU, C., "Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications", *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pp. 653–662, 2005.

[11] VIEIRA, M. A. M., *BEAN: Uma Plataforma Computacional para Rede de Sensores Sem Fio*, Dissertação de mestrado, Universidade Federal de Minas Gerais, Belo Horizonte, MG, Brasil, Abril 2004.

[12] YIH WAN, C., EISENMAN, S. B., "CODA: Congestion detection and avoidance in sensor networks". In: *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems, SenSys*, pp. 266–279, ACM Press, Novembro 2003.

[13] FLOYD, S., JACOBSON, V., "Random Early Detection Gateways for Congestion Avoidance". In: *IEEE/ACM Transactions on Networking*, v. 1, pp. 397–413, Agosto 1993.

[14] SANKARASUBRAMANIAM, Y., AKAN, O., AKYILDIZ, I., "ESRT: Event-to-Sink Reliable Transport in Wireless Sensor Networks". In: *In Proceedings of MobiHoc 03, ACM*, pp. 177–188, Annapolis, Maryland, USA, Junho 2003.

[15] PEREIRA, M. R., DE AMORIM, C. L., DE CASTRO, M. C. S., "Tutorial sobre Redes de Sensores". In: *Cadernos do IME*, v. 14, p. 15, Instituto de Matemática e Estatística (IME) - UERJ, 2003.

[16] STANN, F., HEIDEMANN, J., "RMST: Reliable Data Transport in Sensor Networks". In: *Proceedings of the First International Workshop on Sensor Net Protocols and Applications*, pp. 102–112, Anchorage, Alaska, USA, 2003.

[17] INTANAGONWIWAT, C., GOVINDAN, R., ESTRIN, D., "Directed diffusion: a scalable and robust communication paradigm for sensor networks". In: *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*, pp. 56–67, ACM Press: New York, NY, USA, 2000.

[18] GIANCOLI, E., JABOUR, F., PEDROZA, A., "Collaborative Transport Control Protocol". In: *Proceedings of IEEE International Conference on Computer and Electrical Engineering*, pp. 373–377, Phuket, Tailândia, Dezembro 2008.

[19] GIANCOLI, E., JABOUR, F., PEDROZA, A., "Protocolo de Transporte Colaborativo para Redes de Sensores sem Fio". In: *IX Workshop de Testes e Tolerância a Falhas do Simpósio Brasileiro de Redes de Computadores - SBRC´2008*, p. 12, Rio de Janeiro, Brasil, Dezembro 2008.

[20] COMER, D. E., *Interligação em Rede com TCP/IP*. 3rd ed., v. 1. Editora Campus, 1998.

[21] "Java", http://java.sun.com, acesso em abril de 2006.

[22] LEVIS, P., MADDEN, S., POLASTRE, J., et al., "TinyOS: An operating system for wireless sensor networks". In: *Ambient Intelligence*, pp. 115–148, Springer-Verlag: New York, NY, USA, 2004.

[23] "TinyOS", http://www.tinyos.net/, acesso em fevereiro de 2009.

[24] LEVIS, P., LEE, N., WELSH, M., et al., ""TOSSIM": Accurate and Scalable Simulation of Entire "TinyOS"Applications", Novembro 2003.

[25] CHEVALLAY, C., VAN, R. E., HALL, D. T. A., "Self-organization Protocols for Wireless Sensor Networks". In: *In Thirty Sixth Conference on Information Sciences and Systems*, 2002.

[26] ROEMER, K., MATTERN, F., "The Design Space of Wireless Sensor Networks", *IEEE Wireless Communications*, v. 11, n. 6, pp. 54–61, Dezembro 2004.

[27] SHAH, R. C., ROY, S., JAIN, S., et al., "Data MULEs: Modeling and Analysis of a Three-tier Architecture for Sparse Sensor Networks", *Ad Hoc Networks*, v. 1, n. 2-3, pp. 215–233, Setembro 2003.

[28] LOUREIRO, A. A. F., NOGUEIRA, J. M. S., RUIZ, L. B., et al., "Redes de Sensores Sem Fio", *XXI SBRC*, pp. 179–226, 2003, Brasil.

[29] RUIZ, L. B., CORREIA, L. H. A., VIEIRA, L. F. M., et al., "Arquiteturas para Redes de Sensores Sem Fio", *XXII SBRC*, Maio 2004, Brasil.

[30] HAENSELMANN, T., "Wireless Sensor Network textbook", http://www.informatik.uni-mannheim.de/~haensel/sn_book, acesso em abril 2006.

[31] CHIANG, M. W., ZILIC, Z., CHENARD, J.-S., et al., "Architectures of Increased Availability Wireless Sensor Network Nodes". In: *ITC '04: Proceedings of the International Test Conference on International Test Conference*, pp. 1232–1241, IEEE Computer Society: Washington, DC, USA, 2004.

[32] TILAK, S., ABU-GHAZALEH, N. B., HEINZELMAN, W., "Infrastructure tradeoffs for sensor networks". In: *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pp. 49–58, ACM Press: New York, NY, USA, 2002.

[33] PINTO, A., *Mecanismo de Agregação de Dados Baseado em Técnicas Paramétricas Aplicado em Redes de Sensores*, Dissertação de Mestrado, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil, 2004.

[34] JABOUR, F., GIANCOLI, E., PEDROZA, A., "Redes de sensores móveis: análise da velocidade, comunicação e esforço computacional". In: *7th International Information and Telecommunication Technologies Symposium - I2TS*, p. 8, Foz do Iguaçu, Brasil, Dezembro 2008.

[35] JABOUR, F., GIANCOLI, E., PEDROZA, A., "Mobility support for wireless sensor networks". In: *Proceedings of IEEE International Conference on Computer and Electrical Engineering*, pp. 630–634, Phuket, Tailândia, Dezembro 2008.

[36] JABOUR, F., GIANCOLI, E., PEDROZA, A., "Mobility support for wireless sensor networks". In: *3rd IEEE European Conference on Smart Sensing and Context (EuroSSC)*, p. 1, Outubro 2008.

[37] JABOUR, F., GIANCOLI, E., PEDROZA, A., "Um esquema em duas camadas para suporte à mobilidade em redes de sensores sem fio". In: *XXV Simpósio Brasileiro de Telecomunicações, SBRT*, p. 12, Recife, Brasil, Setembro 2007.

[38] ZIMMERMANN, H., "OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection", *Communications, IEEE Transactions on [legacy, pre - 1988]*, v. 28, n. 4, pp. 425–432, 1980.

[39] YE, W., HEIDEMANN, J., ESTRIN, D., "An Energy-Efficient MAC Protocol for Wireless Sensor Networks". In: *Proc. of International Annual Joint Conference of the IEEE Computer and Communications Societies*, pp. 1567–1576, Junho 2002.

[40] WOO, A., "A transmission control scheme for media access in sensor networks". In: *ACM/IEEE Mobicom Conference*, pp. 221–235, 2001.

[41] BRAGINSKY, D., ESTRIN, D., "Rumor Routing Algorithm for Sensor Networks". In: *Proceedings of the First Workshop on Sensor Networks and Applications (WSNA)*, pp. 22–31, Atlanta, GA, Setembro 2002.

[42] KIM, Y. J., GOVINDAN, R., KARP, B., et al., "CLDP Robust Planarization for Geographic Face Routing". In: *Center for Embedded Network Sensing*, p. 136, Janeiro 2005.

[43] IYER, R., KLEINROCK, L., "QoS Control For Sensor Networks". In: *Proceedings of the IEEE International Conference on Communications (ICC´03)*, v. 26, pp. 517–521, Maio 2003.

[44] YU, Y., GOVINDAN, R., ESTRIN, D., *Geographical and energy aware routing: A recursive data dissemination protocol for wireless sensor networks*, Relatório técnico, UCLA Computer Science Dept., Maio 2001.

[45] AGUAYO, D., BICKET, J., MORRIS, R., *SrcRR: A High Throughput Routing Protocol for 802.11 Mesh Networks (DRAFT)*, Relatório técnico, MIT, 2005.

[46] AKYILDIZ, I. F., SU, W., SANKARASUBRAMANIAM, Y., et al., "Wireless Sensor Networks: A Survey", *Computer Networks*, v. 38, pp. 393–422, 2002.

[47] WANG, C., SOHRABY, K., LI, B., et al., "Issues of transport control protocols for wireless sensor networks". In: *Proceedings of International Conference on Communications, Circuits and Systems*, v. 1, pp. 422–426, Arkansas Univ., Fayetteville, AR, USA, Maio 2005.

[48] ALLMAN, M., PAXSON, V., STEVENS, W., "TCP Congestion Control", RFC 2581 (Proposed Standard), Abril 1999, atualizada pela RFC 3390.

[49] RAGHAVENDRA, C. S., KUMAR, V. K. P., HARIRI, S., "Reliability Analysis in Distributed Systems", *IEEE Trans. Comput.*, v. 37, n. 3, pp. 352–358, 1988.

[50] HEIMLICHER, S., BAUMANN, R., MAY, M., et al., "SaFT: Reliable Transport in Mobile Networks". In: *IEEE International Conference on Mobile Adhoc and Sensor Systems (MASS)*, pp. 477–480, IEEE MASS 2006: Vancouver B.C., Canada, Outubro 2006.

[51] IYER, Y. G., GANDHAM, S., VENKATESAN, S., "STCP: A Generic Transport Layer Protocol for Wireless Sensor Networks". In: *Proceedings of 14th International Conference on Computer Communications and Networks*, pp. 449–454, Houston, TX, USA, 2005.

[52] POSTEL, J., "User Datagram Protocol", RFC 768 (Standard), Agosto 1980.

[53] TANENBAUM, A. S., *Redes de Computadores*. trad. 4 ed. ed. Elsevier: Rio de Janeiro, 2003.

[54] BAKSHI, B. S., KRISHNA, P., VAIDYA, N. H., et al., "Improving Performance of TCP over Wireless Networks". In: *International Conference on Distributed Computing Systems*, p. 1, 1997.

[55] WAN, C.-Y., CAMPBELL, A. T., KRISHNAMURTHY, L., "Pump-Slowly, Fetch-Quickly (PSFQ): A Reliable Transport Protocol for Sensor Networks". In: *IEEE Journal on Selected Areas in Communications*, v. 23, n. 4, pp. 862–872, Atlanta, Georgia, USA, 2005.

[56] VASSIS, D., KORMENTZAS, G., ROUSKAS, A. N., et al., "The IEEE 802.11g standard fo high data rate WLANs", *IEEE Network*, v. 19, n. 3, pp. 21–26, 2005.

[57] COUTO, D. S. J., *High-Throughput Routing for Multi-Hop Wireless Network*, Tese de Doutorado, Massachusetts Institute of Technology, 2004.

[58] CAMPISTA, M. E. M., *Um Novo Protocolo de Roteamento para Redes em Malha sem Fio*, Tese de Doutorado, GTA, COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, Brasil, Dezembro 2008.

[59] KUROSE, J. F., ROSS, K. W., *Redes de Computadores e a Internet: Uma abordagem top-down*. Trad. 3 ed. ed. Addison Wesley: São Paulo, Brasil, 2006.

[60] GIANCOLI, E., JABOUR, F., PEDROZA, A., "CTCP: Reliable Transport Control Protocol for Sensor Networks". In: *2008 Fourth International Conference on Intelligent Sensors, Sensor Nertworks and Information Processing*, p. 10, Sydney, Australia, Dezembro 2008.

[61] RISO, B. G., *Engenharia de Protocolo com LOTOS/Iso*. 1st ed. UFSC, 2004.

[62] NIELSEN, M., PLOTKIN, G., , et al., *Petri Nets, Event Structures and Domains, Part I, Vol. 13*. Theoretical Computer Science, 1981.

[63] MAZIERO, C. A., "ARP - Analisador de Rede Petri", http://www.ppgia.pucpr.br/ maziero/diversos/petri/arp.html, 1990, acesso em dezembro de 2008.

[64] MILNER, R., "A Calculus of Communicating Systems", *Lecture Notes in Computer Science*, v. 92, pp. 123–154, 1980.

[65] SMITH, H., FINGAR, P., "A page about the pi calculus (and Business Process Management)", http://www.fairdene.com/picalculus/, acesso em junho de 2007.

[66] MILNER, R., PARROW, J., WALKER, D., "A calculus of mobile processes, parts", *I and II. Information and Computation*, v. 100, pp. 1–77, 1992.

[67] MOLLER, F., STEVENS, P., "Edinburgh Concurrency Workbench User Manual (Version 7.1)", http://homepages.inf.ed.ac.uk/perdita/cwb/, acesso em dezembro de 2008.

[68] KAFURA, D., "The Concurrency WorkBench (CWB)", http://ei.cs.vt.edu/ cs5204/fall99/ccs.html, acesso em junho de 2007.

[69] SPIEGEL, M. R., *Probabilidade e Estatística*. Primeira ed. *Coleção Schaum*, Makron Books, 1978, Capítulo 1.

[70] SPIEGEL, M. R., *Estatística*. Terceira ed. *Coleção Schaum*, Makron Books, 1993, Capítulo 6.

[71] SEYMOUR LIPSCHUTZ, P., *Probabilidade*. Quarta ed. *Coleção Schaum*, Makron Books, 1993, Capítulos 3 e 4.

[72] "Java Platform, Standard Edition 6 API Specification", http://java.sun.com/javase/6/docs/api/, acesso em janeiro de 2009.

[73] GANESAN, D., KRISHNAMACHARI, B., WOO, A., et al., *An empirical study of epidemic algorithms in large scale multihop wireless networks*, Relatório técnico, University of Massachusetts, Amherst, 2002.

[74] CROSSBOW, "Mica2 Series", http://www.xbow.com, acesso em junho de 2008.

[75] SENTILLA, "moteiv: wireless sensor networks", http://www.sentilla.com/moteiv-transition.html, acesso em junho de 2008.

[76] ENGINEERING, C., LABORATORY, E. Z. N., "Btnodes - a distributed environment for prototyping ad hoc networks", http://btnode.ethz.ch/, acesso em junho de 2008.

[77] OF ENGINEERING, D., SCIENCES, H. U. A., "Codeblue: Wireless sensor networks for medical care", http://www.eecs.harvard.edu/ mdw/proj/codeblue/, acesso em junho de 2008.

[78] GAY, D., WELSH, M., LEVIS, P., et al., "The nesC language: A holistic approach to networked embedded systems", http://www.tinyos.net/papers/nesc.pdf, 2003.

[79] LEVIS, P., "Ad hoc routing component architecture", http://www.tinyos.net/tinyos-1.x/doc/ad-hoc.pdf, acesso em fevereiro 2003.

[80] GBURZYNSKI, P., KAMINSKA, B., OLESINSKI, W., "A tiny and efficient wireless ad-hoc protocol for low-cost sensor networks". In: *DATE '07: Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 1557–1562, EDA Consortium: Nice, France, 2007.

[81] DEMORACSKI, L., "Fault-Tolerant Beacon Vector Routing for Mobile Ad Hoc Networks". In: *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 16*, p. 8, IEEE Computer Society: Washington, DC, USA, 2005.

[82] "Simulating TinyOS Networks", http://www.cs.berkeley.edu/˜ pal/research/-tossim.html, acesso em junho de 2007.

[83] PRESSMAN, R., *Engenharia de Software*. McGraw-Hill Interamericana do Brasil, 2005.

[84] GIANCOLI, E., JABOUR, F., PEDROZA, A., "CTCP: Analisando a Entrega, Latência e Consumo de Energia". In: *7th International Information and Telecommunication Technologies Symposium - I2TS*, p. 8, Foz do Iguaçu, Brasil, Dezembro 2008.

[85] HEINZELMAN, W. R., CHANDRAKASAN, A., BALAKRISHNAN, H., "Energy-Efficient Communication Protocol for Wireless Microsensor Networks". In: *Proceedings of the 33rd Annual Hawaii International Conference*, v. 2, p. 10, IEEE Computer Society: Washington, DC, USA, 2000.

[86] KIM, S., FONSECA, R., DUTTA, P., et al., "Flush: a reliable bulk transport protocol for multihop wireless networks". In: *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pp. 351–365, ACM: Sydney, Australia, 2007.

[87] GIANCOLI, E., JABOUR, F., PEDROZA, A. C. P., ""CTCP: Analisando a Entrega, Latência e Consumo de Energia"", Submissão como Special Issue à IEEE América Latina, Fevereiro 2009, Artigo Premiado: Second Best Paper in I2TS 2008.

[88] GIANCOLI, E., JABOUR, F., PEDROZA, A. C. P., "Collaborative Transport Control Protocol for Wireless Sensor Networks", Submetido a Telecommunication Systems, Fevereiro 2009.

Apêndice A

Códigos do Simulador Estatístico

Na tela chamada **Janela Principal**, temos um menu de seleção que permite executar a classe com as rotinas de cálculo baseadas nas fórmulas dos modelos estatísticos (ACK fim-a-fim ou conf. 1) (janela chamada **Fórmulas**, à direita na figura TAL) ou executar a classe que implementa o **Simulador Estatístico** (ACK fim-a-fim, conf. 1 ou conf. 2) (janela na parte de baixo da referida figura). Esta última, inclusive, apresenta uma saída típica do simulador. Na figura A.1, podemos ver uma instância de execução do simulador utilizado no capítulo 4.

Anexo A.1: Código Fonte do Simulador Estatístico

```
/*
 * SimuladorEstatistico.java
 */
package probCUmECDois;

import java.awt.event.KeyEvent;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Date;
import javax.swing.JFileChooser;
import javax.swing.JOptionPane;

/**
 * @author  Eugenia Jabour
 */
public class SimuladorEstatistico extends javax.swing.JFrame {

    String versao = "1.0.1", cabecalhoLogEmArquivo, ifGeradorDoLoopDaConfDois;
    byte h, R, id, conf, auxEntregasConfZero, numeroDeExecucoes;
    long rodadas, entregas, entregasComFalhasEmAcks, naoEntregas, totalDeEntregasDeACKUm,
         totalDeEntregasDeACKDois, totalDeEntregasDeMSG, totalDeFalhasDeNos;
    float p, q, f;
    No nos[];
    File diretorioCorrente = null;
    FileWriter escreveArquivo = null;

    /** Creates new form SimuladorEstatistico */
    public SimuladorEstatistico() {
```

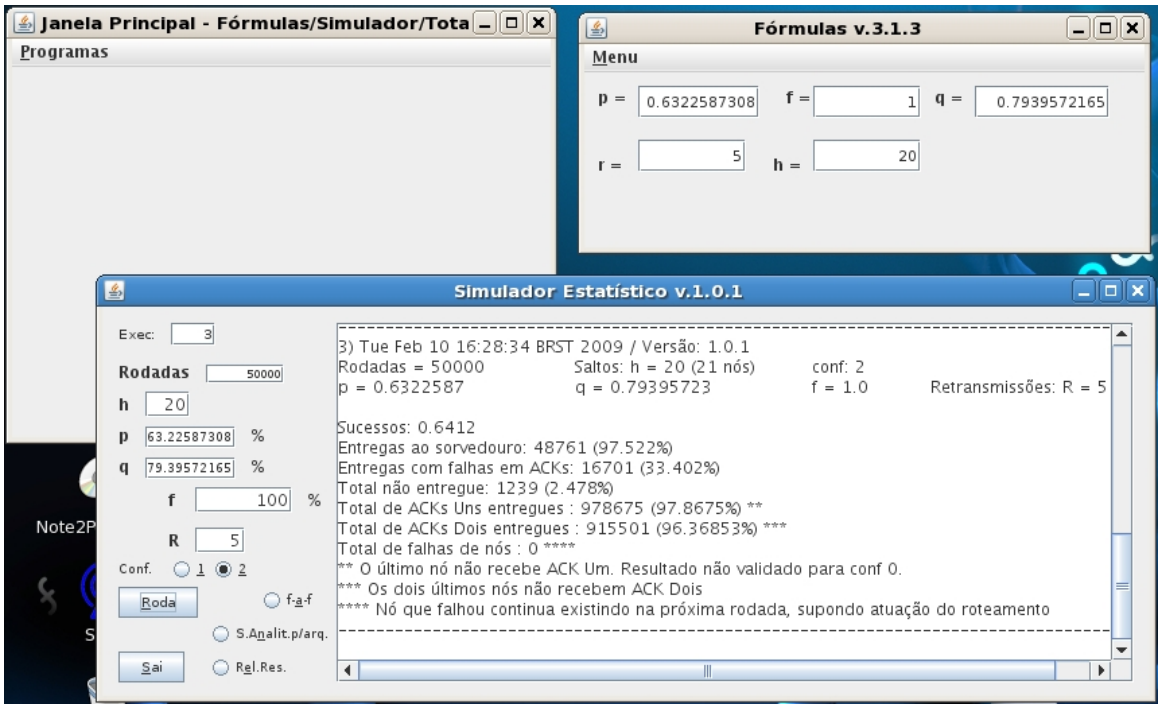

Figura A.1: Execução do Simulador

```
    initComponents();
    this.setLocation(60, 100);
    this.setSize(900, 356);
    this.setTitle(this.getTitle() + versao);
    jRadioButton1.setSelected(true);
    jTextField5.grabFocus();
    jRadioButton4.setSelected(true); }

private void initComponents() {
            //CRIA O FORMULÁRIO DE ENTRADA DE DADOS. O CÓDIGO FOI SUPRIMIDO
            //ENCONTRE O CÓDIGO COMPLETO EM http://www.gta.ufrj.br/~eugenia/
}

private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
    this.dispose(); }

private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    roda(); }

//UMA SÉRIE DE TRATAMENTOS DE EVENTOS DE NAVEGAÇÃO PELO FORMULÁRIO
//FORAM SUPRIMIDOS. ENCONTRE O CÓDIGO COMPLETO EM
//http://www.gta.ufrj.br/~eugenia/

private void roda() {
    if (!validaForm()) {
        JOptionPane.showMessageDialog(this, "Erro no formulário!");
        return; }
    for (int numExec = 1; numExec <= numeroDeExecucoes; numExec++) {
        entregas = 0;
        entregasComFalhasEmAcks = 0;
        naoEntregas = 0;
        totalDeEntregasDeACKUm = 0;
        totalDeEntregasDeACKDois = 0;
        totalDeEntregasDeMSG = 0;
        totalDeFalhasDeNos = 0;
        long umPorCento = rodadas / 100, andamento = 1;
```

```
        System.out.print("\n" + numExec + ") ");
        for (int i = 0; i < rodadas; i++) {
            if (jRadioButton5.isSelected()) {
                inicializaArquivoDeSaida(numExec + "." + i); }
            if (i > (andamento * umPorCento)) {
                if (andamento % 10 == 0) {
                    System.out.print("\n"); }
                System.out.print(h + "/" + andamento + "% ");
                andamento++; }
            criaNos();
            executaUmaRodada();
            limpaNos();
            if (jRadioButton5.isSelected()) {
                try {
                    escreveArquivo.close();
                } catch (IOException ex) {
                    JOptionPane.showMessageDialog(this, "Erro de E/S");
                    ex.printStackTrace(); } } }
        if (jRadioButton4.isSelected()) {
            msg(numExec + ") ");
            imprimeRelatorioFinalResumido();
        } else {
            msg(numExec + ") ");
            imprimeRelatorioFinal(); } }
    System.out.print("\nAcabou . . .\n"); }

private void criaNos() {
    if (jRadioButton1.isSelected()) {
        conf = 1; }
    if (jRadioButton2.isSelected()) {
        conf = 2; }
    if (jRadioButton3.isSelected()) {
        conf = 0; }
    for (byte i = 0; i <= h; i++) {
        nos[i] = new No(i, conf, escreveArquivo); }
    nos[0].setMsg(true);}

private void limpaNos() {
    nos = null;
    nos = new No[h + 1]; }

private void zeraFalhas() {
    for (byte i = 0; i <= h; i++) {
        nos[i].setEmFalha(false); } }

private void executaUmaRodada() {
    int i;
    auxEntregasConfZero = 0;
    if (conf != 0) {
        for (i = 0; i < h; i++) {
            zeraFalhas();
            if (conf == 1) {
                if ((!nos[i].executaProtocolo(conf, R, p, q, f, nos) & !nos[i + 1].temMsg()) |
                        (nos[i + 1].isEmFalha())) {
                    escreveLinha();
                    break; }
            } else { //conf2
                if (i == h - 1) {
                    ifGeradorDoLoopDaConfDois = "0|"+i;
                    nos[i].executaProtocolo((byte)1, R, p, q, f, nos);
                    escreveLinha();
                    break;
                } else {
                    if (!nos[i].executaProtocolo(conf, R, p, q, f, nos) & !nos[i + 1].temMsg() &
                            !nos[i + 2].temMsg()) {
                        ifGeradorDoLoopDaConfDois = "1|"+i;
```

```
                escreveLinha();
                break; }
            if (nos[i + 1].isEmFalha() & nos[i + 2].isEmFalha()) {
                ifGeradorDoLoopDaConfDois = "2|"+i;
                escreveLinha();
                break; }
            if (nos[i].isAckDoisOk() & nos[i + 1].isAckUmOk()) {
                i++;
                if (i == h - 1) {
                    ifGeradorDoLoopDaConfDois = "3|"+i;
                    nos[i].executaProtocolo((byte)1, R, p, q, f, nos);
                    escreveLinha();
                    break;
                } else {
                    ifGeradorDoLoopDaConfDois = "4|"+i;
                    escreveLinha();
                    continue; } } }
            ifGeradorDoLoopDaConfDois = "5|"+i; }
        escreveLinha(); }
    } else { //conf 0
        for (i = 0; i <= R; i++) {
            int j;
            zeraFalhas();
            nos[h].setMsg(false);
            for (j = 0; j < h; j++) {
                if (((float) Math.random()) > f) {
                    totalDeFalhasDeNos++;
                    escreveLinha();
                    break; }
                if (!nos[j].envia('M', p, q, nos)) {
                    escreveLinha();
                    break; } }
            if (j == h) {
                escreveLinha(); }
            if (nos[h].temMsg()) {
                auxEntregasConfZero = 1;
                for (j = h; j > 0; j--) {
                    if (((float) Math.random()) > f) {
                        totalDeFalhasDeNos++;
                        escreveLinha();
                        break; }
                    if (!nos[j].envia('A', p, q, nos)) {
                        escreveLinha();
                        break;
                    } else {
                        nos[j - 1].setMsg(false);
                        escreveLinha(); } }
                if (j == 0) {
                    escreveLinha(); } }
            if (nos[0].isAckUmOk()) {
                break; } } }
    fazTotalizacoes(i); }

private void fazTotalizacoes(int i) {
    if (nos[h].temMsg()) {
        if (conf == 1) {
            entregas++;
            for (int j = 0; j < h; j++) {
                if (!nos[j].isAckUmOk()) {
                    entregasComFalhasEmAcks++;
                    break; } } }
        if (conf == 2) {
            entregas++;
            for (int j = 0; j < (h - 1); j++) {
                if (!nos[j].isAckDoisOk()) {
                    entregasComFalhasEmAcks++;
```

```
                break; } } }
        if (conf == 0) {
            entregas += auxEntregasConfZero;
            if (!nos[0].isAckUmOk()) {
                entregasComFalhasEmAcks++; } }
    } else {
        naoEntregas++; }
    for (int j = 0; j <= h; j++) {
        if (nos[j].isAckUmOk()) {
            totalDeEntregasDeACKUm++; }
        if (nos[j].isAckDoisOk()) {
            totalDeEntregasDeACKDois++; }
        if (nos[j].isEmFalha()) {
            totalDeFalhasDeNos++; } } }

private void imprimeRelatorioFinalResumido() {
    float sucessosPerc = 100 * (((float) entregas - (float) entregasComFalhasEmAcks) / (float) rodadas);
    String stringSucessosPerc = Float.toString(sucessosPerc);
    stringSucessosPerc = stringSucessosPerc.replace('.', ',');
    float entregasPerc = 100 * ((float) entregas / (float) rodadas);
    String stringEntregasPerc = Float.toString(entregasPerc);
    stringEntregasPerc = stringEntregasPerc.replace('.', ',');
    msg("Rodadas=" + rodadas + ";h=" + h + ";conf:" + conf +
        ";p=" + p + ";q=" + q + ";f=" + f + ";R=" + R);
    msg("\nSucessos: " + stringSucessosPerc + " %;");
    msg("\tEntregas: " + stringEntregasPerc + " %");
    msg("\t(v." + versao + ")");
    msg("\n------------------------------------------------------------------------\n"); }

private void imprimeRelatorioFinal() {
    float sucessos = (((float) entregas - (float) entregasComFalhasEmAcks) / (float) rodadas);
    float entregasPerc = 100 * ((float) entregas / (float) rodadas);
    float entregasComFalhasEmAcksPerc = 100 * ((float) entregasComFalhasEmAcks / (float) rodadas);
    float naoEntregasPerc = 100 * ((float) naoEntregas / (float) rodadas);
    float totalDeEntregasDeACKUmPerc = 100 * ((float) totalDeEntregasDeACKUm / (float) (rodadas * h));
    float totalDeEntregasDeACKDoisPerc = 100 * ((float) totalDeEntregasDeACKDois / (float) (rodadas * (h - 1)));

    msg(new Date().toString() + " / Versão: " + versao);
    msg("\nRodadas = " + rodadas + "\tSaltos: h = " + h + " (" + (h + 1) + " nós)\tconf: " + conf +
        "\np = " + p + "\tq = " + q + "\tf = " + f + "\tRetransmissões: R = " + R);
    msg("\n\nSucessos: " + sucessos);
    msg("\nEntregas ao sorvedouro: " + entregas + " (" + entregasPerc + "%)");
    msg("\nEntregas com falhas em ACKs: " + entregasComFalhasEmAcks + " (" +
        entregasComFalhasEmAcksPerc + "%)");
    msg("\nTotal não entregue: " + naoEntregas + " (" + naoEntregasPerc + "%)");
    msg("\nTotal de ACKs Uns entregues : " + totalDeEntregasDeACKUm +
        " (" + totalDeEntregasDeACKUmPerc + "%) **");
    msg("\nTotal de ACKs Dois entregues : " + totalDeEntregasDeACKDois +
        " (" + totalDeEntregasDeACKDoisPerc + "%) ***");
    msg("\nTotal de falhas de nós : " + totalDeFalhasDeNos + " ****");
    msg("\n** O último nó não recebe ACK Um. Resultado não validado para conf 0.");
    msg("\n*** Os dois últimos nós não recebem ACK Dois");
    msg("\n**** Nó que falhou continua existindo na próxima rodada, supondo atuação do roteamento");
    msg("\n------------------------------------------------------------------------------------------------------------\n"); }

private void msg(String m) {
    jTextArea1.append(m);
    jTextArea1.setCaretPosition(jTextArea1.getText().length()); }

private boolean validaForm() {
    try {
        h = Byte.parseByte(jTextField1.getText());
        p = Float.parseFloat(jTextField2.getText());
        p /= 100F;
        q = Float.parseFloat(jTextField3.getText());
        q /= 100F;
```

```
            f = Float.parseFloat(jTextField6.getText());
            f /= 100F;
            R = Byte.parseByte(jTextField4.getText());
            numeroDeExecucoes = Byte.parseByte(jTextField7.getText());
            rodadas = Long.parseLong(jTextField5.getText());
        } catch (NumberFormatException nfe) {
            jTextField7.grabFocus();
            return false; }
        if (rodadas < 1 | R < 0 | h < 1 | p < 0 | p > 100 | q < 0 | q > 100 | f < 0 | f > 100 |
                numeroDeExecucoes < 1 | numeroDeExecucoes > 10) {
            return false;
        } else {
            nos = new No[h + 1];
            cabecalhoLogEmArquivo = " |";
            for (int m = 0; m <= h; m++) {
                cabecalhoLogEmArquivo += String.valueOf(m) + "|"; } }
        return true; }

    private void inicializaArquivoDeSaida(String argNumExecMaisRodada) {
        String nomeArq = argNumExecMaisRodada + "-p" + p + "-f" + f + "-q" + q + "-r" + R + "-h" + h +
                "-v" + versao + ".txt";
        JFileChooser janelaDefineNomeDoArquivo = new JFileChooser();
        janelaDefineNomeDoArquivo.setFileSelectionMode(JFileChooser.FILES_ONLY);
        janelaDefineNomeDoArquivo.setDialogTitle(this.getTitle() + " : Nome");
        if (diretorioCorrente != null) {
            janelaDefineNomeDoArquivo.setCurrentDirectory(diretorioCorrente); }
        if (nomeArq != null) {
            janelaDefineNomeDoArquivo.setSelectedFile(new File(nomeArq)); }
        int result = janelaDefineNomeDoArquivo.showSaveDialog(this);
        if (janelaDefineNomeDoArquivo.getSelectedFile() != null && result !=
                JFileChooser.CANCEL_OPTION) {
            nomeArq = janelaDefineNomeDoArquivo.getSelectedFile().toString();
            File arquivo = new File(nomeArq);
            diretorioCorrente = janelaDefineNomeDoArquivo.getCurrentDirectory();
            if (arquivo.exists()) {
                result = JOptionPane.showOptionDialog(this, "Arquivo " +
                        "jÃ¡ existe. Sobrescrever?", "Cuidado!",
                        JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE,
                        null, null, null); }
            if (result == JOptionPane.YES_OPTION) {
                arquivo.delete();
                try {
                    escreveArquivo = new FileWriter(arquivo);
                    escreveArquivo.write(cabecalhoLogEmArquivo + "\n");
                } catch (IOException ex) {
                    JOptionPane.showMessageDialog(this, "Erro de E/S");
                    ex.printStackTrace(); } } } }

    private void escreveLinha() {
        if (!jRadioButton5.isSelected()) {
            return; }
        String linha;
        for (int k = 0; k <= h; k++) {
            if (k == 0) {
                linha = "|";
            } else {
                linha = ""; }
            if (nos[k].temMsg()) {
                linha += "M";
            } else {
                linha += " "; }
            if (nos[k].isAckUmOk()) {
                linha += "1";
            } else {
                linha += " "; }
            if (nos[k].isAckDoisOk()) {
```

```
                linha += "2";
            } else {
                linha += " "; }
            if (nos[k].isEmFalha()) {
                linha += "f|";
            } else {
                linha += " |"; }
            try {
                escreveArquivo.write(linha);
            } catch (IOException ex) {
                JOptionPane.showMessageDialog(this, "Erro de E/S");
                ex.printStackTrace(); } }
        try {
            escreveArquivo.write(ifGeradorDoLoopDaConfDois + "\n");
        } catch (IOException ex) {
            JOptionPane.showMessageDialog(this, "Erro de E/S");
            ex.printStackTrace(); } }

    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new SimuladorEstatistico().setVisible(true); } }); }
    // Variables declaration - do not modify//GEN-BEGIN:variables
    private javax.swing.ButtonGroup buttonGroup1;
    private javax.swing.JButton jButton1;
    private javax.swing.JButton jButton2;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JLabel jLabel10;
    private javax.swing.JLabel jLabel11;
    private javax.swing.JLabel jLabel2;
    private javax.swing.JLabel jLabel3;
    private javax.swing.JLabel jLabel4;
    private javax.swing.JLabel jLabel5;
    private javax.swing.JLabel jLabel6;
    private javax.swing.JLabel jLabel7;
    private javax.swing.JLabel jLabel8;
    private javax.swing.JLabel jLabel9;
    private javax.swing.JRadioButton jRadioButton1;
    private javax.swing.JRadioButton jRadioButton2;
    private javax.swing.JRadioButton jRadioButton3;
    private javax.swing.JRadioButton jRadioButton4;
    private javax.swing.JRadioButton jRadioButton5;
    private javax.swing.JScrollPane jScrollPane1;
    private javax.swing.JTextArea jTextArea1;
    private javax.swing.JTextField jTextField1;
    private javax.swing.JTextField jTextField2;
    private javax.swing.JTextField jTextField3;
    private javax.swing.JTextField jTextField4;
    private javax.swing.JTextField jTextField5;
    private javax.swing.JTextField jTextField6;
    private javax.swing.JTextField jTextField7;
    // End of variables declaration//GEN-END:variables
}
```

Anexo A.2: Código Fonte da Classe Nó

```
package probCUmECDois;

import java.io.FileWriter;
import java.io.IOException;

/**
 *
 * @author Eugenia Jabour
 */
public class No {
```

```
private byte id, conf;
private boolean msg, ackUmOk, ackDoisOk, emFalha;
FileWriter fw;
byte ackEsperado = 1;

No(byte id, byte conf, FileWriter fw) {
   this.id = id;
   this.conf = conf;
   this.setAckUmOk(false);
   this.setMsg(false);
   this.setAckDoisOk(false);
   this.setEmFalha(false);
   this.fw = fw;
}

boolean envia(char tipo, float p, float q, No[] nos) {
   switch (tipo) {
      case 'M':   //msg conf 1, conf 2 e FaF
         if ((float) Math.random() <= p) {
            nos[getId() + 1].setMsg(true);
            if (ackEsperado == 1) {
               this.setAckUmOk(false);
            } else {
               this.setAckDoisOk(false); }
            escreve("sM");
            return true;
         } else {
            escreve("fM");
            return false; }
      case 'A':   //ack conf1 e ack1 conf2 e FaF
         if ((float) Math.random() <= q) {
            escreve("sA");
            nos[getId() - 1].setAckUmOk(true);
            if (conf == 2) { // this.getId() > 0 já é testado abaixo
               if (nos[getId() - 1].envia('B', p, q, nos)) {
                  return true;
               } else {
                  return false; } }
            return true;
         } else {
            escreve("fA");
            return false; }
      case 'B':   //ack2 conf2
         if (getId() > 0) {
            if ((float) Math.random() <= q) {
               nos[getId() - 1].setAckDoisOk(true);
               escreve("sB");
               return true;
            } else {
               escreve("fB");
               return false; } }
         return true;
      default:
         return false; } }

boolean executaProtocolo(byte conf, byte R, float p, float q, float f, No[] nos) {
   if (conf == 1) {
      return executaProtocoloConfUm(R, p, q, f, nos);
   } else {
      return executaProtocoloConfDois(R, p, q, f, nos); } }

private boolean executaProtocoloConfUm(byte R, float p, float q, float f, No[] nos) {
   for (int i = 0; i <= R; i++) {
      escreve("R1-" + i);
      if (envia('M', p, q, nos)) {
```

```
            if (((float) Math.random()) <= f) {
               if (nos[getId() + 1].envia('A', p, q, nos)) {
                  return true; }
            } else {
               nos[getId() + 1].setEmFalha(true);
               escreve("fN");
               return false;  //f->falha no intervalo crítico. MSG perdida p/ sempre. Não continua no R.
            } } }
      return false; }

   private boolean executaProtocoloConfDois(byte R, float p, float q, float f, No[] nos) {
      for (int i = 0; i <= R; i++) {
         ackEsperado = 2;
         escreve("R2-" + i);
         if (executaProtocoloConfUm((byte) 0, p, q, f, nos)) {
            ackEsperado = 1;
            if (nos[getId() + 1].executaProtocoloConfUm(R, p, q, f, nos)) {
               return true; } } }
      return false; }

   public boolean isAckUmOk() {
      return ackUmOk; }

   public void setAckUmOk(boolean ackUmOk) {
      this.ackUmOk = ackUmOk;
      if (conf == 1) {
         this.setMsg(false); } }

   public boolean temMsg() {
      return msg; }

   public void setMsg(boolean recebeuMSG) {
      this.msg = recebeuMSG; }

   public boolean isAckDoisOk() {
      return ackDoisOk; }

   public void setAckDoisOk(boolean ackDoisOk) {
      this.ackDoisOk = ackDoisOk;
      this.setMsg(false); }

   public byte getId() {
      return id; }

   public byte getConf() {
      return conf; }

   public boolean isEmFalha() {
      return emFalha; }

   public void setEmFalha(boolean emFalha) {
      this.emFalha = emFalha; }

   private void escreve(String s) {
      if (fw ==  null) {
         return; }
      if ( s.startsWith("fN")) {
         s += "[" + (this.getId()+1) + "];"; }
      else {
         s += "[" + this.getId() + "];"; }
      try {
         fw.write(s);
         fw.flush();
      } catch (IOException ex) {
         System.out.println("Erro fw.write(\"(" + s + ")\");"); } }
}
```

```
import java.awt.event.KeyEvent;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Date;
import javax.swing.JFileChooser;
import javax.swing.JOptionPane;

/**
 *
 * @author  Eugenia Jabour
 */

public class PorFormula extends javax.swing.JFrame {

    String versao = "3.1.3";

    /** Creates new form PorFormula */
    public PorFormula() {
        initComponents();
        this.setTitle(this.getTitle() + versao);
    }

    private void initComponents() {
                //CRIA O FORMULÁRIO DE ENTRADA DE DADOS. O CÓDIGO FOI SUPRIMIDO
                //ENCONTRE O CÓDIGO COMPLETO EM http://www.gta.ufrj.br/~eugenia/
     }

     private void jMenuItem1ActionPerformed(java.awt.event.ActionEvent evt) {
         this.dispose();
     }

    private void geraPSporH(FileWriter escreveArquivo) {
        double p, f, q, PS_C1, PS_C1Nova, PS_FaF_C1, PS_FaF_C1Nova, PS_FaF_1ACK;
        int r, h;
        try {
            p = Double.parseDouble(jTFp.getText());
            f = Double.parseDouble(jTFf.getText());
            q = Double.parseDouble(jTFq.getText());
            r = Integer.parseInt(jTFr.getText());
            h = Integer.parseInt(jTFh.getText());
        } catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog(this, "Erro nos dados informados");
            jTFp.grabFocus();
            return;
        }
        String linha = "";
        linha = "Saltos PS_FaF_C1 PS_FaF_1ACK PS_FaF_C1Nova\n";
        try {
            escreveArquivo.write(new Date().toString() + " / Versao: " + versao+"\n");
            escreveArquivo.write("h="+h+"\tr="+r+"\tp="+p+"\tq="+q+"\tf="+f+"\n");
            escreveArquivo.write(linha);
        } catch (IOException ex) {
            JOptionPane.showMessageDialog(this, "Erro de saída de dados - " +
                  "Código PorFormula_escreveArquivo.write()");
            ex.printStackTrace();
        }
        for (int i = 3; i <= h; i++) {
            PS_C1Nova = 0;
            PS_C1 = 1 - (Math.pow((1 - p * f * q), (r + 1)));

            for (int j = 0 ; j <= r ; j++) {
                PS_C1Nova += p*f*q * Math.pow((1-p)+p*f*(1-q), j);
            }
            PS_FaF_C1Nova = Math.pow(PS_C1Nova, i);
```

```
            PS_FaF_C1 = Math.pow(PS_C1, i);

            PS_FaF_1ACK = 1-(Math.pow( (1 - Math.pow(p,i)*Math.pow(q,i)*Math.pow(f,(2*i))), (r + 1)));

            linha = i + " " + PS_FaF_C1 + " " + PS_FaF_1ACK + " " + PS_FaF_C1Nova + "\n";
            try {
               escreveArquivo.write(linha);
            } catch (IOException ex) {
               JOptionPane.showMessageDialog(this, "Erro de saída de dados - " +
                     "Código PorFormula_escreveArquivo.write()");
               ex.printStackTrace();
            }
         }
         JOptionPane.showMessageDialog(this, "Processamento concluído");
      }

      public static void main(String args[]) {
         java.awt.EventQueue.invokeLater(new Runnable() {

            public void run() {
               PorFormula porFormula = new PorFormula();
               porFormula.setLocation(100, 100);
               porFormula.setVisible(true);
            }
         });
      }
      File diretorioCorrente = null;
}
```

Apêndice B

Códigos da Implementação do CTCP

Anexo B.1: Código Fonte do CTCP com Nível 2 de Confiabilidade

```
includes CTCP;
includes CTCPCmd;
includes ACK;
includes TosTime;

module CTCP {
  provides {
    interface StdControl;  }
  uses {
    interface ADC;
    interface Timer;
    //Um temporizador para cada posicao do buffer de transmissao
    interface Timer as Temp_0;
    interface Timer as Temp_1;
    interface Timer as Temp_2;
    interface Timer as Temp_3;
    interface Timer as Temp_4;
    interface Timer as Temp_5;
    interface Timer as Temp_6;
    interface Timer as Temp_7;
    interface Timer as Temp_8;
    interface Timer as Temp_9;
    interface Timer as Temp_Inicial;
    interface Timer as Temp_Mata_no;
    interface Leds;
    interface StdControl as Sounder;
    interface Send;
    interface SendMsg as SendACK;
    interface Receive as Bcast;
    interface ReceiveMsg as RecACK;
    interface RouteControl;
    interface Intercept; }}

implementation {
  enum {
    TAM_VET = 100,
    NUM_MSG = 17,                          //numero de mensagens geradas
    TAMANHO_DO_BUFFER = 10,
    TEMPO_INICIAL= 1024*170,               //Tempo necessário para o roteamento convergir
```

```
    RETRANS_PERMIT = 10,                    //número de retransmissões
    RTT_INICIAL = 47,                       //Valor inicial do temporizador
    NUM_NODES = 25,
    END_SOURCE = 0x0018,                    //25 nos
    //END_SOURCE = 0x0030,                  //49 nos
    //END_SOURCE = 0x0063,  };              //100 nos //atenção ao numero de nós acima
/*****************************************************************/
  float alfa=0.7;
  float beta=1.1;
  float gama=1.5;
/*****************************************************************/
  float alfa2=0.1;
  float beta2=1.3;
  float gama2=1.5;
/*****************************************************************/
  int contaperc=1;                          //conta quantas vezes o temporizador mata_no vai estourar
  bool rebroadcast_adc_packet;
  TOS_Msg gMsgBuffer;
  norace uint16_t gSensorData;              // protegida por gfSendBusy flag
  bool gfSendBusy;
  TOS_Msg teste;
  TOS_Msg mACK2;
  ACKMsg *pACK2;
  CTCPsg msg_retrans;                       //mensagem recuperada do buffer para ser retransmitida
  TOS_Msg gMsgBuffer2;                      //Para usar o get_buffer na retransmissão
  CTCPsg BufferTransm[TAMANHO_DO_BUFFER];   //buffer de transmissão da camada de transporte
  int livre=0;                              //primeira posição livre no vetor
  int guardaretransREINIC[TAMANHO_DO_BUFFER];  //armazena quantas vezes esta msg foi retransmitida
  int guardanumseqACK2[TAM_VET];            //armazena as mensagens que já receberam ACK2
  int guardaACK1[TAMANHO_DO_BUFFER];        //guarda se a msg já recebeu ack1 ou não
  float hora_geracao[TAMANHO_DO_BUFFER];    //guarda a hora em que a msg foi gerada
  float hora_ack[TAMANHO_DO_BUFFER];        //guarda a hora em que seu ACK1 chegou
  float hora_ack2[TAMANHO_DO_BUFFER];       //guarda a hora em que seu ACK2 chegou
  int guardatemp1ou2[TAMANHO_DO_BUFFER];    //guarda se o temporizador foi iniciado com RTT1 ou RTT2
  int guardaretranstemp2[TAMANHO_DO_BUFFER];//guarda quantas vezes uma msg foi retrans por causa do ack2
  int guardaretranstemp1[TAMANHO_DO_BUFFER];//guarda quantas vezes uma msg foi retrans por causa do ack1
  int vet_log[NUM_NODES];
  int indice_buffer=0;
  uint32_t RTT=47;
  uint32_t RTT2=102;
  char timeBuf[128];                        //hora em que o ACK chega
  char timeBuf2[128];                       //hora em que chega o ACK2
  float amostra;
  float amostra2;
  float RTT_prev=0.045898;
  float RTT_prev2=0.1;
  float timeout;
  float segparaint;
  uint16_t ns;
  ACKMsg *paux;
  uint16_t NumSeq;
  int estounobuffer=0;
  int jarecebiack2=0;
  int uva=0;
  int v=0;
  CTCPsg *pmsg_transp;                      //guarda mensagem interceptada
  uint16_t no_origem=999;
  uint16_t ult_intercept;
  uint16_t id_reading;                      //numero de sequencia do pacote
  int posicao, posicaoAnterior;
  float ordemDeGrandeza;
  int horas, minutos, segundos;
  float instanteEnvio;
/*****************************************************************
* Initialization
*****************************************************************/
```

```
static void initialize() {
        uint16_t i;
        timer_rate = INITIAL_TIMER_RATE;
        atomic gfSendBusy = FALSE;
        sleeping = FALSE;
        rebroadcast_adc_packet = FALSE;
        focused = FALSE;
        ns=rand();
        for (i=0; i<TAMANHO_DO_BUFFER; i++) {
                guardaACK1[i]=0;
                guardaretranstemp1[i]=0;
                guardaretranstemp2[i]=0; }
        for (i=0; i<NUM_MSG; i++) {
                ArmazenaAmostra[i]=0;
                ArmazenaAmostra2[i]=0;
                if (i<NUM_MSG-1){
                        ArmazenaTimeout2[i+1]=0; }}}
/*****************************************************************
* Inicia o temporizador
*****************************************************************/
 void Inicia_temp(int aux, int difACK) {
        guardatemp1ou2[aux]=difACK;  //armazeno se o temp foi iniciado com RTT1 ou RTT2
        if (aux==0){
                 call Temp_0.stop();
                 if (difACK == 1) {
                                call Temp_0.start(TIMER_ONE_SHOT, RTT); }
                 if (difACK == 2) {
                                call Temp_0.start(TIMER_ONE_SHOT, RTT2); } }
        .
        .
        .
        if (aux1==9){
                call Temp_0.stop();
                 if (difACK == 1) {
                                call Temp_9.start(TIMER_ONE_SHOT, RTT); }
                 if (difACK == 2) {
                                call Temp_9.start(TIMER_ONE_SHOT, RTT2); } }
        return; }
/*****************************************************************
* Armazena
*****************************************************************/
int Armazena(CTCPsg *pCTCP) {
        int aux2=0;
        int i;
        CTCPsg tempS=(CTCPsg)*pCTCP;
        for(i=0;i<TAMANHO_DO_BUFFER;i++) {
                if (tempS.numseq == BufferTransm[i].numseq) {
                        estounobuffer=1;
                        aux2=i;}}
        if (estounobuffer!=1) {
                if (livre < TAMANHO_DO_BUFFER)  {
                        atomic{
                                BufferTransm[livre]=(CTCPsg)*pCTCP;
                        guardaretransREINIC[livre]= 1;
                        guardaretranstemp1[livre]=0;
                        guardaretranstemp2[livre]=0;
                                aux2=livre;
                                livre++; }} else {
                        livre=0;
                                BufferTransm[livre]=(CTCPsg)*pCTCP;
                                guardaretransREINIC[livre]= 1;
                        guardaretranstemp1[livre]=0;
                        guardaretranstemp2[livre]=0;
                        aux2=livre;
                                livre++; }}}
        return (aux2); }
```

```
/*****************************************************************
* Calcula RTT_prev
*****************************************************************/
float Calcula_RTT_prev(int di, int paraqualRTT){
        if(paraqualRTT == 1) {
                indice++;
                amostra = hora_ack[di] - hora_geracao[di];
                RTT_prev = (alfa*RTT_prev)+((1-alfa)*amostra);
                timeout = beta * RTT_prev; }
        if(paraqualRTT == 2) {
                indice2++;
                amostra2 = hora_ack2[di] - hora_geracao[di];
                RTT_prev2 = (alfa2*RTT_prev2)+((1-alfa2)*amostra2);
                timeout = beta2 * RTT_prev2; }
        return timeout; }
/*****************************************************************
* Tratando os ACKs
*****************************************************************/
void TrataACK(){
        int i;
        CTCPsg temp;
        jarecebiack2=0;
        //verifica se o ACK2 já chegou para esta mensagem
        for (i=0; i<TAM_VET ; i++) {
                if (guardanumseqACK2[i]==NumSeq) {
                        jarecebiack2=1; }}
        //Recebi o ACK1, se a msg não foi gerada por mim enviar ACK2
        if (paux->type ==1) {
                cont_ACK1_rec++;
                if (jarecebiack2 == 0) {
                        for (i=0; i < TAMANHO_DO_BUFFER ; i++) {
                                temp = BufferTransm[i];
                                if (temp.numseq == NumSeq) {
                                        guardaACK1[i]++;
                                if (guardaretranstemp1[i]==0){
                                                //hora em que o ACK chegou
                                                printTime(timeBuf, 128);
                                                hora_ack[i]=Altera_tempo(timeBuf);
                                                atomic{segparaint = 1024*Calcula_RTT_prev(i,1);}
                                                RTT=(uint32_t)segparaint;}
                                                guardaretransREINIC[i]=1;
                                        if (temp.origemaddr==TOS_LOCAL_ADDRESS) {
                                                Para_temp(i);
                                                Inicia_temp(i, 2); } else{
                                                Para_temp(i);  //para o temp pois o ack1 chegou
                                                Inicia_temp(i,2);
                                                //Monta o pacote do ACK2
                                                pACK2->recnumseq = NumSeq;
                                                pACK2->type = 2;
                                                pACK2->origemdoack=TOS_LOCAL_ADDRESS;
                if (call SendACK.send(BufferTransm[i].interceptant, sizeof(ACKMsg), &mACK2) ==SUCCESS) {
                        atomic gfSendBusy = FALSE; } }
                        i=11; }}}}
        //Recebi o ACK2
        if (paux->type ==2) {
                cont_ACK2_rec++;
                if (jarecebiack2 == 0) {
                        atomic{
                                guardanumseqACK2[uva]=NumSeq;  //guardaamsgque recebeu o ack 2
                                uva++;
                                for (i=0; i < TAMANHO_DO_BUFFER ; i++) {
                                        if (BufferTransm[i].numseq == NumSeq) {
                                                Para_temp(i);
                                                guardaretransREINIC[i]=0;
                                                if (guardaretranstemp2[i]==0){
                                                        printTime(timeBuf2, 128);
```

```
                                                hora_ack2[i]=Altera_tempo(timeBuf2);
                                                atomic {segparaint = 1024*Calcula_RTT_prev(i,2);}
                                                RTT2=(uint32_t)segparaint;  }
                                        if (guardaACK1[i]==0) {
                                                pACK2->recnumseq = NumSeq;
                                                pACK2->type = 2;
                                                pACK2->origemdoack=TOS_LOCAL_ADDRESS;
                                                nack2++;
                if (call SendACK.send(BufferTransm[i].interceptant, sizeof(ACKMsg), &mACK2) ==SUCCESS) {
                        atomic gfSendBusy = FALSE; }}
                        i=(TAMANHO_DO_BUFFER+1); }}}}}
        jarecebiack2=0; }
/*****************************************************************
* Trata as mensagens interceptadas
*****************************************************************/
void TrataIntercepta(){
        ACKMsg *pACK = (ACKMsg *)teste.data;
        //Monta o pacote do ACK1
        teste.type = AM_ACKMSG;  //seta o tipo de mensagem no cabeçalho TOS_Msg
        teste.addr = ult_intercept;
        teste.length = sizeof(teste.data);
        pACK->recnumseq = id_reading ;
        pACK->type = 1;
        pACK->origemdoack= TOS_LOCAL_ADDRESS;
        nack++;
        //Envia o ACK1 para o nó de origem
        if (call SendACK.send(ult_intercept, sizeof(ACKMsg), &teste) == SUCCESS) {
                atomic gfSendBusy = FALSE;}
        // Se for o nó zero ele tera que mandar um ACK1 e um ACk2 ao mesmo tempo
        if (TOS_LOCAL_ADDRESS == 0x0000) {
                atomic gfSendBusy = TRUE;
                teste.type = AM_ACKMSG;  //seta o tipo de mensagem no cabeçalho TOS_Msg
                teste.addr = ult_intercept;
                teste.length = sizeof(teste.data);
                pACK->recnumseq = id_reading ;
                pACK->type = 2;
                pACK->origemdoack= TOS_LOCAL_ADDRESS;
                if (call SendACK.send(ult_intercept, sizeof(ACKMsg), &teste) == SUCCESS) {
                        atomic gfSendBusy = FALSE; }}
                cont_intercept++; }
/*****************************************************************
* Envia mensagem
*****************************************************************/
task void SendData() {
        CTCPsg *pReading;
        uint16_t Len;
        //Criando uma mensagem de transporte para enviar.
        if (pReading = ((CTCPsg *)(call Send.getBuffer(&gMsgBuffer,&Len)))) {
        pReading->type = CTCP_TYPE_SENSORREADING;
        pReading->parentaddr = call RouteControl.getParent();
              pReading->reading = gSensorData;
              pReading->origemaddr = TOS_LOCAL_ADDRESS;
              pReading->interceptaddr = TOS_LOCAL_ADDRESS;
              pReading->interceptant = TOS_LOCAL_ADDRESS;
              pReading->numseq = ns;
        ns++;    //numero de sequencia
        cont++;  //numero de msg criadas pelo nó corrente
                //Armazenando a mensagem no buffer de transporte antes de enviar
                indice_buffer=Armazena(pReading);
                //hora em que a msg foi gerada
        printTime(timeBuf, 128);
                hora_geracao[indice_buffer]=Altera_tempo(timeBuf);
                //Enviando a msg já armazenada em buffer
                if ((call Send.send(&gMsgBuffer,sizeof(CTCPsg))) == SUCCESS) {
                        atomic gfSendBusy = FALSE;     //esta linha nao existe na conf1
                //Iniciar o temporizador a cada msg enviada menos o no zero
```

```
                if (TOS_LOCAL_ADDRESS != 0x0000) {
                        Inicia_temp(indice_buffer, 1); } } else {
                atomic gfSendBusy = FALSE; }}}
/*************************************************************************
* Retransmite
*************************************************************************/
        void Retransmite(int indice_retrans, int estourodequem) {
        uint16_t Len2;
        CTCPsg *pReading2;
        int proibido=0;
        //Pegando a msg correspondente ao indice_retrans
        msg_retrans = BufferTransm[indice_retrans];
        //Se esta restransmissao foi gerada por um estouro do RTT2 o gamma não deve ser usado
        if (estourodequem==1) {
                guardaretranstemp1[indice_retrans]++;
                RTT = gama*RTT;}
        if (estourodequem==2) {
                //Alterando o valor do timeout por causa do algoritmo de karn
                guardaretranstemp2[indice_retrans]++;
                RTT2 = gama*RTT2;}
        //Verificando número de vezes que a msg foi retransmitida
        if (guardaretransREINIC[indice_retrans]<RETRANS_PERMIT) {
                guardaretransREINIC[indice_retrans] = guardaretransREINIC[indice_retrans]+1;
                } else {proibido=1;}
        //retransmitindo
        if (proibido !=1) {
                if (pReading2 = (CTCPsg *)(call Send.getBuffer(&gMsgBuffer2,&Len2))) {
                        pReading2->type = msg_retrans.type;
                pReading2->parentaddr = call RouteControl.getParent();
                pReading2->reading = msg_retrans.reading;
                pReading2->origemaddr = msg_retrans.origemaddr;
                pReading2->numseq = msg_retrans.numseq;
                        pReading2->interceptaddr = msg_retrans.interceptaddr;
                        pReading2->interceptant = msg_retrans.interceptant;
                        cont_retrans++;
                        if ((call Send.send(&gMsgBuffer2,sizeof(CTCPsg))) == SUCCESS) {
                                atomic gfSendBusy = FALSE;
                                Inicia_temp(indice_retrans, 1); } else {
                                        atomic gfSendBusy = FALSE; } else {
                                        dbg(DBG_USR3, Erro"); }}
        proibido=0;
        atomic gfSendBusy = FALSE;
        return; }
/*************************************************************************/
  command result_t StdControl.init() {
        initialize();
        return SUCCESS;}
  command result_t StdControl.start() {
        return call Temp_Inicial.start(TIMER_ONE_SHOT, TEMPO_INICIAL); }
  command result_t StdControl.stop() {
        return call Timer.stop(); }
/*************************************************************************
* Eventos
*************************************************************************/
event result_t Timer.fired() {
        if (cont<15){ //para enviar 15 msg cont vai de zero a 14
                if (TOS_LOCAL_ADDRESS == END_SOURCE) {
                        call ADC.getData(); }}
return SUCCESS;}
/*************************************************************************
* Temporizador inicial
*************************************************************************/
event result_t Temp_Inicial.fired() {
        return call Timer.start(TIMER_REPEAT, timer_rate); }
/*************************************************************************/
* TEMPORIZADORES
```

```
/*****************************************************************/
  event result_t Temp_0.fired() {
        atomic {
                if (!gfSendBusy) {
                        indice_buffer=0;
                        gfSendBusy = TRUE;
                        if (guardatemp1ou2[indice_buffer]==1){
                                Retransmite(indice_buffer, 1); }
                        if (guardatemp1ou2[indice_buffer]==2){
                                Retransmite(indice_buffer, 2); } }}
        return SUCCESS; }
/*****************************************************************/
.
.
.
/*****************************************************************/

 event result_t Temp_9.fired() {
        atomic {
                if (!gfSendBusy) {
                        indice_buffer=9;
                        gfSendBusy = TRUE;
                        if (guardatemp1ou2[indice_buffer]==1){
                                Retransmite(indice_buffer, 1); }
                        if (guardatemp1ou2[indice_buffer]==2){
                                Retransmite(indice_buffer, 2); } }}
        return SUCCESS; }
/*****************************************************************/
  event result_t Send.sendDone(TOS_MsgPtr pMsg, result_t success) {
        atomic gfSendBusy = FALSE;
        return SUCCESS; }
/*****************************************************************/
event result_t SendACK.sendDone(TOS_MsgPtr pMsg, result_t success) {
        atomic gfSendBusy = FALSE;
        return SUCCESS; }
/*************************************************************/
* Interceptando uma mensagem
/*************************************************************/
event result_t Intercept.intercept(TOS_MsgPtr msg, void* payload, uint16_t payloadLen) {
        pmsg_transp = (CTCPsg *)payload;
        printTime(timeBuf, 128);
        //lê o nó origem da mensagem e o num de sequencia
        no_origem = pmsg_transp->origemaddr;
        id_reading = pmsg_transp->numseq;
        ult_intercept = pmsg_transp->interceptaddr;
        //Altera o campo interceptaddr da msg interceptada
        pmsg_transp->interceptant = pmsg_transp->interceptaddr;
        pmsg_transp->interceptaddr = TOS_LOCAL_ADDRESS;
        //Inicia a tarefa que vai gerar os ACKs
        TrataIntercepta();
        //Guarda a mensagem interceptada no cache
        indice_buffer=Armazena(pmsg_transp);
        if (estounobuffer==1) {
                atomic estounobuffer=0;
                return FAIL; } else {//esta linha faz com que a mensagem não seja reencaminhada
                if (TOS_LOCAL_ADDRESS != 0x0000){
                        Inicia_temp(indice_buffer, 1);}
                printTime(timeBuf, 128);
                hora_geracao[indice_buffer]=Altera_tempo(timeBuf);
                return SUCCESS; }} //esta linha faz com que a mensagem seja reencaminhada
/*************************************************************/
* Preciso receber os ACKs
/*************************************************************/
event TOS_MsgPtr RecACK.receive(TOS_MsgPtr pMsg) {
        if (pMsg->addr==TOS_LOCAL_ADDRESS) {     //o ack é enderecado para este no
                atomic {
```

```
                        pACK2 = (ACKMsg *)mACK2.data;
                        paux = (ACKMsg *)pMsg->data;
                        NumSeq = paux->recnumseq;
                        TrataACK(); }}
        return pMsg; }
}
```

Anexo B.2: Código Fonte do CTCP com Nível 1 de Confiabilidade e Controle de Congestionamento

```
includes CTCP;
includes CTCPCmd;
includes ACK;
includes TosTime;
module CTCPM {
  provides {
    interface StdControl;}
  uses {
    interface ADC;
    interface Timer;
    //Um temporizador para cada posicao do buffer de transmissao
    interface Timer as Temp_0;
    interface Timer as Temp_1;
    interface Timer as Temp_2;
    interface Timer as Temp_3;
    interface Timer as Temp_4;
    interface Timer as Temp_5;
    interface Timer as Temp_6;
    interface Timer as Temp_7;
    interface Timer as Temp_8;
    interface Timer as Temp_9;
    interface Timer as Temp_Inicial;
    interface Timer as Temp_START;
    interface Leds;
    interface StdControl as Sounder;
    interface Send;
    interface SendMsg as SendACK;
    interface Receive as Bcast;
    interface ReceiveMsg as RecACK;
    interface RouteControl;
    interface Intercept;
    interface Time;
    interface TimeUtil; }}

implementation {
  enum {
    TAMANHO_DO_BUFFER = 10,
    TAM_VET = 26,                                   //número de mensagens enviadas
    PATAMAR = 6,                                    //pATAMAR
    RETRANS_PERMIT = 10,                            //significa nove retransmissões
    NUM_NODES = 25,
    RTT_INICIAL = 47,                               //tirei do log RTT_3
    END_SOURCE = 0x0018,                            //25 nos
    //END_SOURCE = 0x0030,                          //49 nos
    //END_SOURCE = 0x0063,                          //100 nos //atenção ao numero de nós acima
    //END_SOURCE = 0X0004,};
  int contaperc=1;                                  //conta quantas vezes o temporizador mata_no vai estourar
  bool rebroadcast_adc_packet;
  TOS_Msg gMsgBuffer;
  norace uint16_t gSensorData;                  // protegida por gfSendBusy flag
  bool gfSendBusy;
  TOS_Msg teste;
  CTCPMsg msg_retrans;                              //mensagem recuperada do buffer para ser retransmitida
  TOS_Msg gMsgBuffer2;                              //Para usar o get_buffer na retransmissão
  CTCPMsg BufferTransm[TAMANHO_DO_BUFFER];      //buffer de transmissão da camada de transporte
```

```
int livre=0;                                        //primeira posição livre no vetor
int quantnumseq[TAMANHO_DO_BUFFER];                 //armazena o instante de envio de cada msg
float hora_geracao[TAMANHO_DO_BUFFER];              //guarda a hora em que a msg foi gerada
float hora_ack[TAMANHO_DO_BUFFER];                  //guarda a hora em que seu ACK chegou
int foiprocessada[TAMANHO_DO_BUFFER];               //indica se a msg foi processada ou não pelo noh
int perda=0;                                        //conta quantas msg foram sobrepostas sem serem processadas
uint16_t origens[TAMANHO_DO_BUFFER];                //guarda endereços de origens contidas no buf
int indice_buffer=0;
uint32_t RTT;
uint16_t ns;
uint16_t ns_1;
uint32_t RTTSTART=1024;
ACKMsg *paux;
uint16_t NumSeq;
int estounobuffer=0;
char timeBuf[128];
int msgprocessadas=0;
int posocupadas=0;
int envioustop=0;
CTCPMsg msg_trans;                                  //mensagem do buffer que vai ser transm pela primeira vez
int statusdoNO=0;
TOS_Msg gMsgBuffer3;
float amostra;
float RTT_prev=0.045898;
float timeout;
float segparaint;
/*************************************************************/
float alfa=0.7;
float beta=1.1;
// float gama=1.5; //usado antes do CC
float gama=1; //usado para CC
/*************************************************************/
int indice=0; //indice dos dois próximos vetores
float ArmazenaAmostra[TAM_VET];
float ArmazenaTimeout[TAM_VET];
uint16_t ArmazenaNumSeq[TAM_VET];
int uva=0;
int posicao, posicaoAnterior;  //posicao (e posicao anterior) em um array (vetor)
float ordemDeGrandeza;         //auxiliar p/ retirar minutos de timeBuf
int horas, minutos, segundos;
float instanteEnvio;
uint16_t vizinhosDevoStart[TAM_VET];  //Vizinhos para onde alguma vez jah
//enviei STOP. Devido ao cenário estático da rede, o vetor acima não é resetado.
int qtdVizinhosDevoStart = 0;
/*************************************************************************
* Initialization
*************************************************************************/
static void initialize() {
  timer_rate = INITIAL_TIMER_RATE;
  atomic gfSendBusy = FALSE;
  sleeping = FALSE;
  rebroadcast_adc_packet = FALSE;
  focused = FALSE;
  RTT=RTT_INICIAL;
  dbg(DBG_USR3, "%d: O valor do RTT ao iniciar o programa eh:%d\n",TOS_LOCAL_ADDRESS, RTT);
  ArmazenaAmostra[0]=0;
  ArmazenaTimeout[0]=RTT;
  ArmazenaNumSeq[0]=0;
  ns_1=rand();
  ns=ns_1;}
/*************************************************************************
* Inicia o temporizador
*************************************************************************/
void Inicia_temp(int aux) {
      if (aux==0){
              call Temp_0.start(TIMER_ONE_SHOT, RTT);}
```

```
      .
        .
        .
        if (aux==9){
                call Temp_9.start(TIMER_ONE_SHOT, RTT); }
        return; }
 /*****************************************************************
 * Pára o temporizador
 *****************************************************************/
 void Para_temp(int aux1) {
        if (aux1==0){
                call Temp_0.stop(); }
        .
        .
        .
       if (aux1==9){
                call Temp_9.stop(); }
       return; }
/*****************************************************************
* Envia ACK
*****************************************************************/
void EnviaACK(int tipo, uint16_t paraquem, uint16_t numS) {
     ACKMsg *pACK = (ACKMsg *)teste.data;
        //Monta o pacote do ACK(1) ou ACK_STOP(3) ou start(4)
        teste.type = AM_ACKMSG;  //seta o tipo de mensagem no cabeçalho TOS_Msg
        teste.addr = paraquem;
     teste.length = sizeof(teste.data);
     pACK->recnumseq = numS;
     if (tipo==1){ pACK->type = 1;}
     if (tipo==3){ pACK->type = 3;}
     if (tipo==4){ pACK->type = 4;}
        nack++; //conta numero de acks transmitidos
        //Envia o ACK para o nó de origem
        if (call SendACK.send(paraquem, sizeof(ACKMsg), &teste) == SUCCESS) {
            atomic gfSendBusy = FALSE; } }
/*****************************************************************
* VerificaBuf
*****************************************************************/
   void VerificaBuf(int quemchamouArmazena, uint16_t endereco, uint16_t sequencia) {
        int i, c, c1;
        int acheinumseqstop=0;
        //quemchamouArmazena pode ser:
        //1 - SendData - criando msg
        //2 - Intercepta msg
        //3 - Recebeu ACK1
        //4 - temporizador
        //msgprocessadas eh o num de mensagens que jah foram processadas e podem ser sobrepostas
        msgprocessadas=0;
        posocupadas=0;
        for(i=0;i<TAMANHO_DO_BUFFER;i++) {
                if (foiprocessada[i]==1) {
                        msgprocessadas=msgprocessadas+1; }
                if (foiprocessada[i]==2) {
                        posocupadas=posocupadas+1; }}
        if (quemchamouArmazena ==1) {
                //quando quemchamouArmazena==1 não posso enviar nenhum
                //tipo de ack pois foi o proprio no que gerou
                dbg(DBG_USR3, "%d: Armazena: Não enviei ACK porque estou CRIANDO\n",TOS_LOCAL_ADDRESS);}
        if (quemchamouArmazena ==2) {
                //Se o num de posicoes ocupadas for maior ou igual ao patamar
                if (posocupadas>=PATAMAR) {
                        dbg(DBG_USR3, "%d: Armazena: ENVIAR STOP\n",TOS_LOCAL_ADDRESS);
                        //enviar stop tipo 3
                        //paro o temporizador do start
                        call Temp_START.stop();
                        //Envia stop, que eh um ACK tipo 3
```

```
                        EnviaACK(3,endereco,sequencia);
                        atomic envioustop=1;
                        //verifica se a origem jah esta no vetor
                        for(i=0;i<qtdVizinhosDevoStart;i++) {
                                if (endereco == vizinhosDevoStart[i]) {
                                        umseqstop=1;
                                        i=qtdVizinhosDevoStart; } }
                                        //guardar no vetor de stop
                                        if(acheinumseqstop==0) {
                                                vizinhosDevoStart[qtdVizinhosDevoStart++]=endereco; }}else {
                                                //se não enviou o stop devo enviar um ack normal
                                EnviaACK(1,endereco,sequencia); } }
        if (quemchamouArmazena ==3) {
                        //quemchamou foi o Temporizador
                        //Se o num de posicoes ocupadas for menor que o patamar envia start
                        if (posocupadas<(PATAMAR-2)) {
                        if (envioustop==1){
                                for(c=0;c<qtdVizinhosDevoStart;c++) {
                                        EnviaACK(4,vizinhosDevoStart[c],sequencia); }
                                call Temp_START.start(TIMER_REPEAT, RTTSTART);
                                atomic envioustop=0; }}}
        if (quemchamouArmazena ==4) {
        //quemchamou foi o ACK1
                //Se o num de posicoes ocupadas for menor que o patamar envia start senão não faz nada
                if (posocupadas<(PATAMAR-2)) {
                        dbg(DBG_USR3, "%d: Armazena: ENVIAR START MOTIVO 4\n",TOS_LOCAL_ADDRESS);
                        //enviar start tipo 4
                        for(c1=0;c1<qtdVizinhosDevoStart;c1++) {
                                EnviaACK(4,vizinhosDevoStart[c1],sequencia); }
                                call Temp_START.start(TIMER_REPEAT, RTTSTART); }
        return; }
/*****************************************************************
* Armazena
*****************************************************************/
int Armazena(CTCPMsg *pCTCP, int quemchamou) {
        int acheiposLivre=0;
        int aux2=0;
        int i;
        CTCPMsg tempS=(CTCPMsg)*pCTCP;
        for(i=0;i<TAMANHO_DO_BUFFER;i++) {
        if (tempS.numseq == BufferTransm[i].numseq) {
                atomic estounobuffer=1;
                  aux2=i;
                  EnviaACK(1, BufferTransm[i].interceptant,BufferTransm[i].numseq); }}
        if (estounobuffer!=1) {
        //se amsg não esta no buffer devo armazenala
             atomic {
                for(i=0;i<TAMANHO_DO_BUFFER;i++) {
                        if (foiprocessada[i]==1 || foiprocessada[i]==0) { //a msg jah foi processada
                                livre=i;
                                i=TAMANHO_DO_BUFFER+1;
                                acheiposLivre=1; }}
                        if (acheiposLivre==0) {
                                livre=0; }
                acheiposLivre=0; }
                if (foiprocessada[livre]==2){
                        perda++; }
                BufferTransm[livre]=(CTCPMsg)*pCTCP;
                quantnumseq[livre]= 1;
                foiprocessada[livre]=2;
                //toda vez que uma nova msg é armazenada ela recebe foiprocessada igual a 2
                if (TOS_LOCAL_ADDRESS == 0000){
                        foiprocessada[livre]=1; }
                aux2=livre;
                VerificaBuf(quemchamou,BufferTransm[aux2].interceptant,BufferTransm[aux2].numseq);
}
```

```
        return (aux2); }
    /*****************************************************************
    * Calcula RTT_prev
    *****************************************************************/
    float Calcula_RTT_prev(int di){
        indice++;
        amostra = hora_ack[di] - hora_geracao[di];
        RTT_prev = (alfa*RTT_prev)+((1-alfa)*amostra);
        timeout = beta * RTT_prev;
        return timeout; }
   /*****************************************************************
   * SendData() - Geracao de msg
   *****************************************************************/
   task void SendData() {
        CTCPMsg *pReading;
        uint16_t Len;
        //Criando uma mensagem de transporte para enviar.
        if (pReading = ((CTCPMsg *)(call Send.getBuffer(&gMsgBuffer,&Len)))) {
        pReading->type = CTCP_TYPE_SENSORREADING;
              pReading->parentaddr = call RouteControl.getParent();
              pReading->reading = gSensorData;
              pReading->origemaddr = TOS_LOCAL_ADDRESS;
              pReading->interceptaddr = TOS_LOCAL_ADDRESS;
              pReading->interceptant = TOS_LOCAL_ADDRESS;
              pReading->numseq = ns;
              pReading->IDconex = ns_1;
              ns++;     //numero de sequencia
        cont++;  //numero de msg criadas pelo nó corrente
        //Armazenando a mensagem no buffer de transporte antes de enviar
        indice_buffer=Armazena(pReading, 1);
         //hora em que a msg foi gerada
        printTime(timeBuf, 128);
                hora_geracao[indice_buffer]=Altera_tempo(timeBuf);
              if (statusdoNO == 0) {
                        if ((call Send.send(&gMsgBuffer,sizeof(CTCPMsg))) == SUCCESS) {
                                if (TOS_LOCAL_ADDRESS != 0000) {
                                        Inicia_temp(indice_buffer); } else {
                                        atomic gfSendBusy = FALSE; } else {
                                        atomic gfSendBusy = FALSE; }}}
   /*****************************************************************
   * Retransmite
   *****************************************************************/
   void Retransmite(int indice_retrans) {
        uint16_t Len2;
        CTCPMsg *pReading2;
        int proibido=0;
        //Pegando a msg correspondente ao indice_retrans
        msg_retrans = BufferTransm[indice_retrans];
        //Alterando o valor do timeout por causa do algoritmo de karn
        RTT = gama*RTT;
        //Verificando número de vezes que a msg foi retransmitida
        if (quantnumseq[indice_retrans]<RETRANS_PERMIT) {
                quantnumseq[indice_retrans] = quantnumseq[indice_retrans]+1; } else {
                proibido=1; }
        if (proibido !=1) {
                if (statusdoNO==0) {
                if (pReading2 = (CTCPMsg *)(call Send.getBuffer(&gMsgBuffer2,&Len2))) {
                        pReading2->type = msg_retrans.type;
                        pReading2->parentaddr = call RouteControl.getParent();
                        pReading2->reading = msg_retrans.reading;
                        pReading2->origemaddr = msg_retrans.origemaddr;
                        pReading2->numseq = msg_retrans.numseq;
                          pReading2->interceptaddr = msg_retrans.interceptaddr;
                          pReading2->interceptant = msg_retrans.interceptant;
                          pReading2->IDconex = msg_retrans.IDconex;
                          cont_retrans++;
```

```
                        if ((call Send.send(&gMsgBuffer2,sizeof(CTCPMsg))) == SUCCESS) {
                              Inicia_temp(indice_retrans); } else {
                             Inicia_temp(indice_retrans);
                                      atomic gfSendBusy = FALSE; } } else {dbg(DBG_USR#, "Erro")
                                      } else { Inicia_temp(indice_retrans); }}
                                  proibido=0;
                                  atomic gfSendBusy = FALSE;
        return; }
 /*************************************************************************
 * EnviaMsgBuf
 *************************************************************************
   void EnviaMsgBuf() {
        int i;
        //Pegando cada msg que deve ser enviada
        for(i=0;i<TAMANHO_DO_BUFFER;i++) {
                if (foiprocessada[i]==2) {
                        Para_temp(i);
                        Inicia_temp(i);
                        msg_trans = BufferTransm[i];
                if (pReading3 = (CTCPMsg *)(call Send.getBuffer(&gMsgBuffer3,&Len3))) {
                        pReading3->type = msg_trans.type;
                        pReading3->parentaddr = call RouteControl.getParent();
                        pReading3->reading = msg_trans.reading;
                        pReading3->origemaddr = msg_trans.origemaddr;
                        pReading3->numseq = msg_trans.numseq;
                        pReading3->interceptaddr = msg_trans.interceptaddr;
                        pReading3->interceptant = msg_trans.interceptant;
                        pReading3->IDconex = msg_trans.IDconex;
                        if ((call Send.send(&gMsgBuffer3,sizeof(CTCPMsg))) == SUCCESS) {
                                //Iniciar o temporizador a cada msg enviada
                        Para_temp(i);
                          Inicia_temp(i); } else {
                        Para_temp(i);
                        Inicia_temp(i);
                                atomic gfSendBusy = FALSE; }} else {
                                        dbg(DBG_USR3, Erro"); } }}
        return; }
//*************************************************************************
  command result_t StdControl.init() {
    initialize();
    return SUCCESS; }
  command result_t StdControl.start() {
    return call Temp_Inicial.start(TIMER_ONE_SHOT, 1024*150);
    return SUCCESS; }
  command result_t StdControl.stop() {
    return SUCCESS; }
/*************************************************************************
* Eventos
*************************************************************************/
  event result_t Timer.fired() {
        if (TOS_LOCAL_ADDRESS == END_SOURCE){
        call ADC.getData(); }
        return SUCCESS; }
/*************************************************************************
* Temporizador inicial
*************************************************************************/
  event result_t Temp_Inicial.fired() {
        call Timer.start(TIMER_REPEAT, timer_rate);
/*************************************************************************/
  event result_t Temp_START.fired() {
        call Temp_START.stop();
        VerificaBuf(4,0,0);
      return SUCCESS; }
/*************************************************************************
* TEMPORIZADORES
*************************************************************************/
```

```
  event result_t Temp_0.fired() {
        atomic {
        if (!gfSendBusy) {
                indice_buffer=0;
                        gfSendBusy = TRUE;
                Retransmite(indice_buffer); } else {
                Para_temp(0);
                Inicia_temp(0); }}
     return SUCCESS; }
/*****************************************************************
.
.
.
*****************************************************************/
 event result_t Temp_9.fired() {
        atomic {
        if (!gfSendBusy) {
                indice_buffer=9;
                        gfSendBusy = TRUE;
                Retransmite(indice_buffer); } else {
                  Para_temp(9);
                        Inicia_temp(9); }}
     return SUCCESS; }
/*****************************************************************/
async event result_t ADC.dataReady(uint16_t data) {
        atomic {
        if (!gfSendBusy) {
                gfSendBusy = TRUE;
                gSensorData = data;
                post SendData(); }}
      return SUCCESS; }
/*****************************************************************/
  event result_t Send.sendDone(TOS_MsgPtr pMsg, result_t success) {
    atomic gfSendBusy = FALSE;
    return SUCCESS;}
/*****************************************************************/
 event result_t SendACK.sendDone(TOS_MsgPtr pMsg, result_t success) {
        //evento que indica que um ACK acabou de ser enviado
        atomic gfSendBusy = FALSE;
        return SUCCESS; }
/*****************************************************************
* Interceptando uma mensagem
*****************************************************************/
event result_t Intercept.intercept(TOS_MsgPtr msg, void* payload, uint16_t payloadLen) {
        CTCPMsg *pmsg_transp = (CTCPMsg *)payload;
        uint16_t no_origem=99;
        uint16_t ult_intercept;
        uint16_t id_reading; //numero de sequencia do pacote de transporte
        //hora em que a msg chegou
        printTime(timeBuf, 128);
        //lê o nó origem da mensagem e o num de sequencia
        atomic gfSendBusy = TRUE;
        no_origem = pmsg_transp->origemaddr;
        id_reading = pmsg_transp->numseq;
        ult_intercept = pmsg_transp->interceptaddr;
        //Altera o campo interceptaddr da msg interceptada
        pmsg_transp->interceptant = ult_intercept;
        pmsg_transp->interceptaddr = TOS_LOCAL_ADDRESS;
        //Guarda a mensagem interceptada no cache e o armazena vai enviar o ack
        indice_buffer=Armazena(pmsg_transp, 2);
        //Somando as msgs no no zero para o log final
        if (TOS_LOCAL_ADDRESS == 0000) {
                vet_log[no_origem]++; }
        cont_intercept++;
        if (estounobuffer==1) {
        atomic estounobuffer=0;
```

```
                return FAIL; } //esta linha faz com que a mensagem não seja reencamin
        if (estounobuffer!=1) {
        if (statusdoNO==0) {
                if (TOS_LOCAL_ADDRESS != 0000){
                        Inicia_temp(indice_buffer); }
                        printTime(timeBuf, 128);
                        hora_geracao[indice_buffer]=Altera_tempo(timeBuf);
                        return SUCCESS; } //esta linha faz com que a mensagem seja reencamin
        if (statusdoNO==3) {
        //impeço a msg de seguir seu caminho. mas ela já esta no buffer
        return FAIL; }}}
/*****************************************************************/
* Preciso receber os ACKs
/*****************************************************************/
  event TOS_MsgPtr RecACK.receive(TOS_MsgPtr pMsg) {
        int i;
      int tipodoack;
      CTCPMsg temp;
      if (pMsg->addr==TOS_LOCAL_ADDRESS) {
                cont_ACK1_rec++;  //conta o nu m de ACK1 recebido
                atomic {
                        paux = (ACKMsg *)pMsg->data;
                        NumSeq = paux->recnumseq;
                        tipodoack= paux->type;
                for (i=0; i < TAMANHO_DO_BUFFER ; i++) {
                temp = BufferTransm[i];
                        if (temp.numseq == NumSeq) {  //A msg está no buffer
                        if (quantnumseq[i]==1){   //A mensagem não foi retransmitida
                                        printTime(timeBuf, 128);
                                        hora_ack[i]=Altera_tempo(timeBuf);
                                        segparaint = 1024*Calcula_RTT_prev(i);
                                        RTT=(uint32_t)segparaint; }
                                foiprocessada[i]=1;
                                VerificaBuf(3,temp.interceptant,temp.numseq);
                                quantnumseq[i]=0;
                                Para_temp(i);
                                i=11; }}
                    if (paux->type==3) {
                        statusdoNO = 3; }
                        if (paux->type==4) {
                        statusdoNO = 0;
                                if (TOS_LOCAL_ADDRESS != 0000){
                                        EnviaMsgBuf(); }}}}
      return pMsg; }


}
```