ROBCAST: A RELIABLE MAC LAYER PROTOCOL FOR BROADCAST IN WIRELESS SENSOR NETWORKS

by

Srivats Balachandran

A thesis

submitted to the Faculty of the Graduate School

of State University of New York at Buffalo

in partial fulfillment of the requirements

for the degree of Master of Science

September 2006

ACKNOWLEDGEMENT

This work would not have been completed without help and support of many individuals. I would like to thank everyone who has helped me along the way. Particularly: Dr. Murat Demirbas for providing me an opportunity to conduct my master's research under him and for his guidance and support over the course of it. Dr. Chunming Qiao for serving on my thesis committee, and valuable suggestions. Muzammil for his invaluable help with the simulations. My present and past roommates at buffalo for all the memorable times. Lastly, my family without whose support none of this would have been possible.

ABSTRACT

Real-world experiments [1, 2] have shown that wireless sensor networks exhibit certain traffic patterns where events of interest have been noticed to cause a burst of activity at nodes propagating the information over the network. However, bursts of activity from multiple transmitters in a neighborhood results in energy wastage due to collisions. Current protocols use RTS-CTS handshakes to avoid collisions and alleviate the hidden terminal problem. However such a solution is neither efficient nor reliable for WSNs. Thus, motivated by the goal of reliable data transmission using minimum power, we propose RoBcast - a round based solution for reliably broadcasting data over a single-hop using information from detected collisions. The link-level reliability of RoBcast can form the building block upon which future applications and protocols can be designed for WSNs.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recent advances in low-power wireless radios and MEMS technology has made it feasible to deploy large-scale wireless ad hoc networks for real-world application scenarios. Several wireless ad hoc networks, specifically wireless sensor networks (WSNs), have been deployed in recent experiments. For example, scientific data has been collected from WSNs deployed, for monitoring the nesting behavior of endangered birds in a remote island [1], for monitoring the temperature and humidity in vineyards [2], for sniper localization [3], and for classification and tracking of trespassers [4, 5].

1.1 Motivation

Such real-world experiments have provided valuable scientific data and has provided us with a better understanding of the working of WSNs. Different communication patterns have been identified. The goal of these experiments is to reliably report data, while consuming the least amount of power [6]. In this paper, we look at the design of RoBcast, a

MAC layer protocol for reliable delivery of messages for the most common communication pattern, broadcast.

Although support for reliable unicast using RTS/CTS handshake has existed traditionally in 802.11 [7] and in sensor network MAC layer protocols [8], there has not been any support for reliable broadcasting. Popular wireless sensor network MAC protocols [6, 8] provide best effort delivery of broadcast packets. However, reliable communication over a single hop is an essential component of the future [9]. Reliable packet delivery is essential in sensor/actuator networks where all nodes need to consistently reach a consensus. For example, robotic highway safety/construction markers [10] have to consistently take correct decisions, otherwise a robot cone that has an inconsistent view of the system could enter in to traffic and create a significant hazard. As another example, sensor/actuator devices coordinating regulator valves in a factory floor may need to take consistent decisions to prevent a malfunction.

A major hurdle for reliable delivery is the hidden node problem, where simultaneous transmissions from two transmitters outside each other's carrier sensing range collide at the receiver node. There have been many studies [11, 12] showing the detrimental effects of hidden node problem. In particular, more than 50% message loss has been reported due to hidden node problem under bursty traffic loads in wireless sensor networks [13]. Current solutions for avoiding the hidden node problem revolve around acknowledgments with RTS/CTS providing a partial solution for unicast transmissions.

Simple extensions of popular unicast protocols using RTS/CTS or acknowledgments [14, 15, 16, 17] run into problems when applied to broadcast messages. Implosions of the CTS and ACK packets at the source decreases efficiency. Performing handshakes on a node-by-node basis similar to unicast can guarantee reliability. However the high

communication overhead and difficulty in maintaining neighbor lists makes such protocols unattractive. Solutions that use separate control channels/frequencies like BTMA [18] are not suited to wireless sensor networks with a small footprint and energy constraints. Many TDMA based approaches have also been proposed that provide reliable transmissions but at the expense of unused bandwidth and energy.

1.2 Contributions

Our contribution is a reliable MAC level layer for broadcast, namely RoBcast, for low power ad hoc wireless sensor networks.

In RoBcast, receiver side carrier sensing based collision detection techniques [19] are used to reduce data loss. All nodes are synchronized to maintain a global synchronization of rounds with each round having three phases: RTS, NCTS and Data phase. If a node j has data to transmit, a request for transmission (RTS) is sent by j during the RTS phase. Neighboring nodes respond to an invalid RTS or collided RTS's by transmitting a not clear to send message (NCTS) during the NCTS phase. The transmitter backs off from transmitting the data for this round if it either receives a NCTS message or detects a collision in the NCTS phase. This avoids potential collision of data during the Data phase, and ensures reliable delivery of the payload within single-hop distance of the transmitting node.

Secondly, RoBcast - a synchronized round based protocol implements controlled sleep periods to improve power efficiency of the system by turning the radio off when possible. Synchronization can be achieved efficiently by using a time synchronization protocol [20, 21], or an ad hoc mechanism based on a sync packet for each round in [8].

In order to compare the performance of RoBcast with other Broadcast protocols, simulations were performed in Prowler [22]. The results are presented in Chapter 5.

With the increasing interest within the research community in wireless sensor networks and numerous protocols being designed, many papers have looked at the current issues in the field [9, 23]. One of the more important problem at hand being the lack of a system architecture for sensor networks. [9] presents such an architecture, the SensorNet Protocol and has identified - single-hop broadcast communication as the "narrow-waist" for WSN's analogous to IP [24] for Internet protocols. We believe that a reliable broadcast service, such as RoBcast, will serve as the building block and provide a platform for the development of future protocols and applications.

1.3 Organization

After the related work section, in Chapter 2 we look at previously published work related to RoBcast. We introduce preliminaries like notation, terminology, model and discuss our program and network model briefly in Chapter 3. We present our RoBcast protocol and a formal proof of correctness in Sections 4.1 and 4.2 respectively. Extensions to the protocol are discussed in Section 4.4. Chapter 5 presents our simulation results that compare performance of RoBcast with other reliable and unreliable broadcast solutions. Experiments performed in the lab using motes are described in Chapter 6. Finally, conclusion and suggestions for future work are in Chapter 7.

Chapter 2

Related Work

In this section we review previous work on reliable broadcast protocols.

In Robust Broadcast[25], the author selects a neighboring node as a collision detector and performs a RTS-CTS handshake to take control of the channel before a broadcast. However, this does not necessarily guarantee that collisions are absent at other nodes. The hidden node problem still remains and may affect other neighboring node. Tang and Gerla[14] proposed a simple extension to IEEE802.11 to support broadcast. The extension incorporates CA and RTS/CTS control frames similar to unicast schemes. A *source* broadcasts RTS after a CA phase and sets the $WAIT\_FOR\_CTS$ timer for back-off. Neighbors not in YIELD state reply with a CTS and set the $WAIT\_FOR\_DATA$ timer. The source node expects a CTS from any of its neighbors before transmitting DATA. Nodes not involved in the broadcast set their state to YIELD if they receive a CTS.

In BSMA[15], the authors extend [14] by incorporating negative acknowledgments (NAK). If neighbors do not receive DATA after transmitting a CTS, the $WAIT\_FOR\_DATA$

expires and they transmit a NAK. The *source* backs off upon receiving a NAK and retransmits DATA .

However, these protocols [14, 15] require DS (direct sequence) capture ability which enables the radio to lock to a sufficiently strong signal in the presence of interfering signals.

In [14], the authors also present a variation for radio's without DS by detecting busy channels when multiple transmitters are active. This information is then used by the *source* to back off before transmitting DATA.

BMW [16] provides reliable broadcast using RTS-CTS-DATA-ACK handshakes with each neighbor individually in round robin while reverting to IEEE802.11 under high channel contention.

BMMM [17] improves BMW by broadcasting the DATA packet only once. The *source* reserves the channel using a RTS-CTS handshake with each of its neighbors, broadcasts the DATA and requests acknowledgments from each neighbor individually.

BMW and BMMM thus deliver the message eventually but with individual transactions with its neighbors. The disadvantages of these schemes is the high latency, control overhead and, maintenance of a neighbor table.

BTMA [18] prevents collisions by maintaining a separate radio channel for control information. A busy tone is transmitted by the receiver if currently receiving data. A *source* transmits data only if the control channel is idle, thus avoiding collisions and hidden terminals. However, energy and space constraints make the implementation of a separate radio/frequency difficult to achieve in wireless sensor networks.

TDMA protocols provide collision free medium access. A network of N nodes demands a schedule of N time slots with a dedicated time slot for each node. Nodes transmit data

during their allocated time slot, hence effectively avoiding collisions. However such a system requires efficient time synchronization for the entire network. Changes in the network topology requires a modification in the schedule or slot allocation. Finally, static allocation of slots can leave many slots unused reducing the throughput of the network. Many protocols have been proposed to improve pure TDMA [26, 27, 28, 29, 30, 31, 32, 33].

The Adaptive Broadcast protocol (ABROAD)[32] is one such TDMA based protocol which integrates CSMA/CA within each time slot in a frame. The nodes are assumed to be capable of determining 0, 1 or multiple transmissions in the channel corresponding to idle, successful reception and a collision respectively. The protocol works with the *source* broadcasting a RTB (Request To Broadcast) in its assigned time slot. Neighbors upon receiving the RTB, reply with a CTB (Clear To Broadcast) thus informing nodes up to 2 hops from *source* of the transmission. If a slot is idle during the sensing interval, other nodes may try to claim the slot by transmitting a RTB. If a collision is detected - nodes reply with a NCTB (Not Clear To Broadcast). Upon detection of no NCTB or collisions, the channel is identified as free and the node that sent a RTB uses the slot, else it defers its transmission to its assigned slot or contends for other idle slots. AGENT [33] improves ABROAD by transmitting a CTS for unicast messages during the CTB/NCTB phase, thus providing a mechanism to deliver both unicast and broadcast messages reliably.

Avoiding the overheads of allocating a time slot for every single node in a schedule designed for the entire network, we can design a round based system that takes advantage of the synchronous nature of TDMA based protocols, while allowing for the flexibility of random-access protocols. RoBcast is a such a protocol, which proves that there is a significant advantage in using a time synchronized, round based transmission schedule. The round based algorithm allows the nodes to select the transmitting node in a random

fashion based on RTS and (N)CTS control messages.

Chapter 3

Preliminaries

In this chapter, we introduce the basic concepts used hereafter.

3.1 Notation & Terminology

Program

A *program* is said to consist of a set of variables and actions at each node. We use the notation $v_i.x$ to denote a program variable x residing at the node v_i. The set of all variables and their values at a given time-point defines the state of a program.

Action

Each action has the form:

$$< guard > \longrightarrow < statement >$$

A *guard* is a boolean expression over variables. An action whose guard evaluates to $TRUE$ in a particular state is said to be *enabled* for that state and is executed.

Statement

An assignment statement updates one or more variables. When a node uses its radio to transmit a message, the message is broadcast over the channel. Hence, we use the terms transmit and broadcast interchangeably and denote a message broadcast statement as $bcast(msg)$. Message transmissions, unless specified explicitly as "unicast", always mean a broadcast. Similarly, successful reception of a message at a node is denoted by the statement $receive(msg)$. However, the receipt of a collision is represented by the statement $receive(\pm)$.

Formulae

A formula $(op\ j : R.j : X.j)$ denotes the value obtained by performing the (commutative and associative) op on the $X.j$ values for all j that satisfy $R.j$. As special cases, where op is conjunction, we write $(\forall j : R.j : X.j)$, and where op is disjunction, we write $(\exists j : R.j : X.j)$. Thus, $(\forall j : R.j : X.j)$ may be read as "if $R.j$ is true then so is $X.j$", and $(\exists j : R.j : X.j)$ may be read as "there exists an j such that both $R.j$ and $X.j$ are true". Where $R.j$ is true, we

omit $R.j$.

3.2 Network Model

We consider a mesh topology that is represented by a graph $G(V,E)$, where V and E are the set of all nodes and links in the network respectively. The network is said to consist of N stationary nodes identified by $v_1, v_2 \ldots, v_N$.

The set of all single hop neighbors of v_i is represented as $Nbr(v_i)$. Any node v_k is said to be a single hop neighbor of a node v_i iff v_k can establish bi-directional links directly with v_i i.e., $< v_i, v_k > \in E$ and $< v_k, v_i > \in E$. It is to be noted that $v_k \notin Nbr(v_k)$.

From this definition, it is follows that any node in the neighborhood of v_i will have v_i in its neighborhood.

$$v_i \in Nbr(v_k) \equiv v_k \in Nbr(v_i)$$

The set of all nodes that can be reached by v_i in two hops is denoted as $Nbr^2(v_i)$ and defined as follows:

$$(\bigcup v_j : v_j \in Nbr(v_i) : (Nbr(v_j) - (Nbr(v_i) \cup v_i)))$$

The network is assumed to be dense enough for any transmitter, v_i, to possess atleast one receiver in its neighborhood $Nbr(v_i)$. Also, neighbors are assumed to always have a common neighbor as represented by the below formula.

$$(\forall v_i, v_j : v_j \in Nbr(v_i) : (Nbr(v_i) \cap Nbr(v_j)) \neq NULL)$$

Radio

Each node possesses an omni directional radio that is half duplex. Its transmit power is maintained constant at all times. The radio is also assumed capable of performing carrier sensing to detect channel activity. These characteristics are satisfied by the commonly used Chipcon radios [34] satisfy these conditions.

Traffic

We assume all traffic consists of broadcast messages, where each message is addressed by default to all nodes in the neighborhood and discuss later, an extension to ensure the reliable delivery of unicast messages.

3.3 Fault Model

We consider a system where faults can occur arbitrarily. Nodes may fail, stop and crash - corrupting the state of a node. Arrival of new nodes or changes to the topology are considered as transient faults. Moreover, the channel might corrupt messages due to collision, fading or interference. We have not attempted to maintain any distinction between the different kinds of failure. With such a fault model, we state that a program is *self-stabilizing* iff after faults stop occurring, the program eventually recovers from a arbitrary state to a state where its specification is satisfied.

3.4 Collision Detection

Since each message is addressed to all nodes in the neighborhood, we treat any loss of data, due to bit errors or collision of transmitted messages, as a collision. Though protocols presented in [32, 33, 35] utilize the information of detected collisions in simulations, there is no previously mentioned attempt at detecting collisions using motes.

Carrier sensing, physical and virtual, has been widely employed in wireless devices to detect and avoid collisions. Traditionally in MAC layers like IEEE 802.11, IEEE 802.15.4, and most wireless sensor network MAC protocols, carrier sensing has been used primarily at the transmitters: If a node wants to transmit *DATA*, it first senses the medium for any activity in the channel, and begins transmission only if there is no activity. This technique however is not very effective, as the collision occurs at the receiver and the physical carrier sensing at the transmitter does not realize the possible collisions at the receiver's radio. Hence the motivation to design RoBcast to sense the medium at the receiver while informing the transmitter of possible collisions.

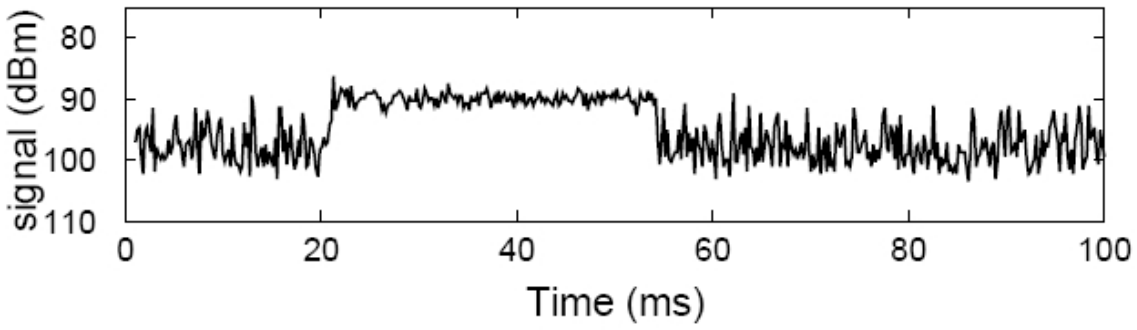

Figure 3.1: The variation in channel energy between noise and data. source:[6].

Using physical carrier sensing, nodes can differentiate between genuine activity, such as a message or collision, and noise based on the variance in the channel energy. When

there is genuine activity on the channel, there is a fairly constant channel energy which stays above the noise floor. Random noise exhibits significant variance in channel energy which can be identified by occasional pits below the noise floor as illustrated in Figure 3.1. Further, we can identify collisions if a node in the idle state (when it is not transmitting or receiving a message, or synchronizing to receive a message) detects, using its carrier sensing mechanism, an intense activity on the medium. In our preliminary experiments with the Mica2 mote platform [34], our sensing mechanism searches for the pits in the idle state and detects genuine activity in the radio if a pit is not found for a long period. We find that our receiver side carrier-sensing based collision detection performs well and detects more than 95% of the collisions accurately.

3.5 Synchronization

The popularity of TDMA protocols and the advantages of maintaining synchronized time on individual nodes has lead to a lot of research on synchronization in a Ad Hoc network. Various protocols [20, 21, 36] have been developed for synchronization in sensor networks. Some protocols like S-MAC [8] utilize a SYNC packet as part of their protocol to synchronize the individual nodes sleep schedules. Real world experiments [3] have proved that it is possible to obtain a fine level of time synchronization using such protocols.

However, round synchronization does not require all the nodes to have a global view of time. It has been noted in [37] that round synchronization can be achieved with gradient synchronization instead of a global synchronization.

RoBcast, could use such a round level synchronization. Given any two nodes v_i and v_j, both nodes are assumed to have the same global view of rounds i.e., rounds start (and end)

at the same time on both nodes as illustrated in Figure 3.2.

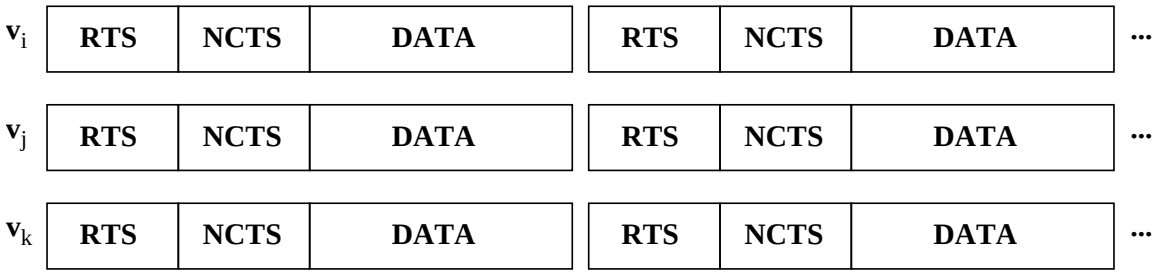

Figure 3.2: Synchronized rounds in RoBcast

Round synchronization for RoBcast can be achieved using different algorithms.

- **Time Synchronization:** Using algorithms defined in [20, 21, 36], we can achieve fine grained time synchronization between all the nodes clocks. Then, after the rounds start, the time synchronization protocol's messages can be transmitted using RoBcast.

- **Beacon:** For a single hop neighborhood, it may be possible to use a initiator node that sends a start message for each round.

- **Ad Hoc Synchronization:** It may be possible to achieve a lightweight ad hoc round synchronization by exploiting the schedule of the messages.

Chapter 4

Protocol

In this section, we present RoBcast, a reliable broadcast protocol and provide a formal proof of correctness, showing that RoBcast eliminates the hidden node problem. We also prove that RoBcast is self-stabilizing in the face of arbitrary state corruptions, and discuss extensions to RoBcast for achieving energy-efficiency and ad hoc, on-demand round-synchronization.

4.1 Protocol Description

Each node v_i maintains a single variable, $status$. $v_i.status$ has a domain of $\{idle,\ candidate,\ transmit,\ veto\}$. As a shorthand, we use $v_i.x$ to denote $v_i.status = x$. $v_i.candidate$ means v_i wants to transmit a message, and, $v_i.transmit$ means v_i has exclusive access to the channel and it will be transmitting the rest of its packets in the DATA phase of consecutive rounds. $v_i.veto$ implies that there exists multiple *candidates* in $Nbr(v_i)$, and v_i will veto the candidates from becoming transmitters. If none of the above holds for v_i, $v_i.idle$ is true by

default. Initially for all v_i, $v_i.status = idle$.

The variable "phase" is an external variable (provided by a round synchronization service), notifying v_i of which phase of the round, RTS, NCTS or DATA, v_i is in. All the nodes have a consistent view of the phase variable due to our round synchronization requirement.

As seen in Figure 4.1, RoBcast consists of six actions as explained below.

Action 1 is enabled in the RTS phase when a node has data to be sent and needs access to the channel. This happens either when an $idle$ node has new data to be transmitted, or when a node is in the $transmit$ state during a multi-part message's transmission. In case the node is transmitting the first message, i.e. $state := idle$, the node transits to $candidate$ state.

Action 2 is enabled in the RTS phase when a node is $idle$ and is not transmitting data in this round. While listening to the channel for activity any collision detected indicates collision of RTS messages. In this case, the node detects the existence of multiple transmitters in its single-hop neighborhood, and goes to $veto$ state to block a $candidate$ from going to $transmit$ state. If an RTS message is successfully received, the number of packets of the data message to be received as part of this transmission is stored in $data\_to\_receive$.

Action 3 is enabled in the $NCTS$ phase when a node is in the $veto$ state. Upon execution, the node broadcasts a veto message in the format of a $NCTS$ message[1].

Action 4 is enabled in the NCTS phase for a node v_i in the $candidate$ state. If v_i receives an $NCTS$ message or detects a collision, v_j backs off from the transmission and transits to $idle$ state.

Action 5 is enabled in the DATA phase when a node v_i's state is $candidate$ or $transmit$. v_i's state is set to $transmit$ and the $DATA$ message is broadcast. If the $data\_to\_send$ field

[1]Switching from transmission to listening is on the order of microseconds, hence the feasibility of this scheme[6].

(1) phase=$RTS \wedge (v_i.data\_to\_send > 0) \wedge (v_i.data\_to\_receive == 0)$

$\longrightarrow$ bcast(RTS)
if ($v_i.idle$)
then $v_i.state := candidate$

[]

(2) phase=$RTS \wedge v_i.idle \wedge (v_i.data\_to\_send == 0 \vee vi.back\_off == TRUE)$

$\longrightarrow$ if (receive($\pm$))
$v_i.state := veto$
else if (receive(msg))
$v_i.data\_to\_receive := msg.data\_to\_receive$

[]

(3) phase=$NCTS \wedge v_i.veto$

$\longrightarrow$ bcast($NCTS$)
$v_i.state := idle$

[]

(4) phase=$NCTS \wedge v_i.candidate \wedge receive(\pm \; or \; msg)$

$\longrightarrow$ $v_i.state := idle$

[]

(5) phase=$DATA \wedge (v_i.candidate \vee v_i.transmit)$

$\longrightarrow$ $v_i.state := transmit$
bcast(msg)
$v_i.data\_to\_send := v_i.data\_to\_send - 1$
if $v_i.data\_to\_send == 0$
then $v_i.status := idle$

[]

(6) phase=$DATA \wedge (v_i.idle \wedge v_i.data\_to\_receive > 0)$

$\longrightarrow$ receive(msg)
if (receive_timeout)
$v_i.data\_to\_receive := 0$
else
$v_i.data\_to\_receive := v_i.data\_to\_receive - 1$

Figure 4.1: Program actions for j.

of the message indicates that all packets as part of this message have been transmitted, v_i transits to *idle* state.

Action 6 is enabled in the DATA phase when a node v_i's state is *idle*. If $data\_to\_receive > 0$ from Action 2, the node expects packets to arrive and receives the part of the message that has been transmitted. However, in case a *candidate* receives an *NCTS* during the *NCTS* phase and backs off via Action 4, a timeout occurs in Action 6 because of the absence of a transmitter. In this case the data is not received by the node and the its status is set to *idle*.

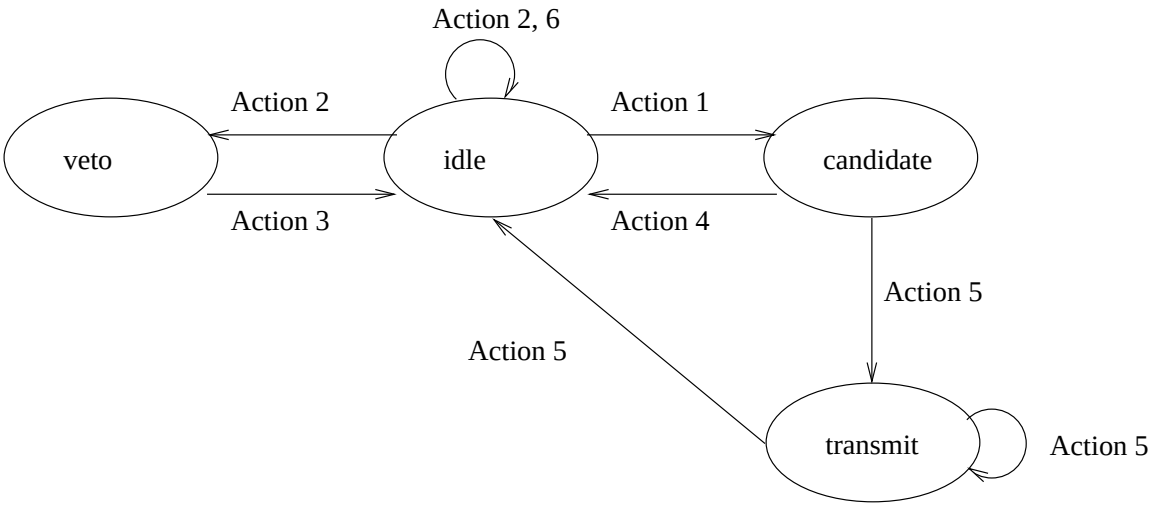

Figure 4.2: The effect of actions on the *status* variable.

Figure 4.2 illustrates the effect of actions on the status variable of a node. Note that Actions 1, 2 are enabled only in the RTS phase, Actions 3, 4 are enabled only in the NCTS phase, and Action 5, 6 are enabled only in the DATA phase.

4.2 Correctness proof

We can now prove Lemma 4.1 about contending nodes. Lemma 4.1 states that if at the beginning of a round's RTS phase, for any node v_i there exists no node in its neighborhood v_j, $v_j \in Nbr(v_i)$, such that $vj.transmit$, then there can exist only one node, v_k, with access to the channel during the *DATA* phase of that round.

Lemma 4.1. *(**Leader election**) If* $(\forall v_i : v_i \in G : (\forall v_j : v_j \in Nbr(v_i) : \neg v_j.transmit))$ *in the beginning of a round, then* $(\forall v_j, v_k : v_j, v_k \in Nbr(v_i) : (v_j.transmit \wedge v_k.transmit) \implies v_j = v_k)$ *in the DATA phase of that round.*

Proof. If at the beginning of a round, none of the neighbors of a node v_i are in the transmit state, they can transit to such a state only after contending for the channel during the *RTS* and *NCTS* phase of that round. Let us consider the situation where v_j and v_k, two neighboring nodes of v_i, contend during the *RTS* phase by transmitting a *RTS* message. This will only result in a collision at the receiver v_i, which will veto both the "*candidates*", v_j and v_k, to *idle* state. Hence, when all nodes are *idle* initially, during the *DATA* phase of that round, if $v_j.transmit$ and $v_k.transmit$ then $v_j = v_k$. □

So starting from a initial state where all the nodes are in *idle* state, it is possible to have only one transmitter in the neighborhood of a particular given node.

If the transmitting node has multiple packets to be transmitted as part of this message, it will retain access over the channel until all the packets are transmitted successfully. In order for this to happen reliably, no other *candidate* must be allowed to go to *transmit* state.

Lemma 4.2 states that if at the beginning of a round, for any node v_i there exists a node $v_j \in Nbr(v_i)$, such that $vj.transmit$, then there will exist only one node, v_j, with access to

the channel during the *DATA* phase of that round.

Lemma 4.2. *(**Leader preservation**) If* $(v_i : v_i \in G : (\exists v_j : v_j \in Nbr(v_i) : v_j.transmit))$ *in the beginning of a round, then* $(\forall v_k : v_k \in Nbr(v_i) : v_k.transmit \implies v_k = v_j)$ *in the DATA phase of the round.*

Proof. If v_j and v_k are two neighboring nodes of v_i such that at the beginning of the round $v_j.transmit$ and $v_k.idle$. During the RTS phase both nodes transmit a RTS message resulting in a collision at the receiver v_i, which will broadcast a *veto* message during the following $NCTS$ phase. On receipt of a *veto* message, v_k transits from *candidate* to *idle* state whereas v_j remains unaffected (since it already has exclusive access to the channel). Hence, during the *DATA* phase, only v_j remains in the *transmit* state within the single-hop neighborhood of v_i in the system. □

We now prove that in RoBcast there exists no hidden node, i.e for any given node, there can exist at most one transmitter in its single-hop neighborhood. Theorem 4.3 states that if $I1$ and $I2$ hold, there can be at most one node within single-hop of a node v_i that has access to the channel in the *DATA* phase of any round.

Theorem 4.3. *(**No hidden node**) Let* $I1$ *denote phase* $=$ *DATA* $\wedge$ $(\forall v_i : v_i \in G : (\forall v_j, v_k : v_j, v_k, \in Nbr(v_i) : v_j.transmit \wedge v_k.transmit \implies v_j = v_k))$. $I1$ *is an invariant of the RoBcast protocol.*

Proof. $I1$ follows from Lemma 4.1 and Lemma 4.2. Lemma 4.1 states that, in the absence of faults, starting from initial states, it is always the case that a node has at-most one neighbor in *transmit* state. Lemma 4.2 states that, in the absence of faults, if there exists a node in the middle of a multi-packet transmission, it will remain the only node in *transmit*

state during its transmission. Hence, when there exists a transmitter, no other *candidate* is allowed to transit to the *transmit* state. That is, I1 is preserved always. □

The absence of hidden nodes ensures that the corruption of transmissions is decreased. However, for a time synchronized system to reliably deliver data, along with the absence of collisions the intended recipients must be ready to receive data.

Lemma 4.4 states that, it is always the case that if there is a node v_i is in *transmit* state, then all nodes in $Nbr(v_i)$ will be *idle* and hence receive data during the *DATA* phase.

Lemma 4.4. *Let* $I2$ *denote phase* $=$ $DATA \wedge (\forall v_i : v_i \in G : (\exists v_j : v_j \in Nbr(v_i) : v_j.transmit) \implies v_i.idle)$. $I2$ *is an invariant of the RoBcast protocol.*

Proof. From Lemma 4.1 and Lemma 4.2 it follows that, in the absence of faults, starting from initial states, it is always the case that a node has at-most one neighbor in *transmit* state. Hence, if there exists a node v_j in *transmit* state in $Nbr(v_i)$, then v_i cannot be in *transmit* state. Also, any node that is in the *veto* or *candidate* state during the *NCTS* phase, cannot remain in the same state at the end of the phase. Thus, $I2$ is preserved and v_i is always in the *idle* state if there exists a one-hop neighbor in *transmit* state. □

4.3 Self-stabilization

As we prove in Lemma 4.3 and Lemma 4.4, in the absence of faults, starting from initial states, $I1$ and $I2$ hold for RoBcast and, hence, from Theorem 4.3 we conclude that RoBcast eliminates the hidden node problem. However, due to faults, such as transient memory corruption, message loss, or changes in network topology, $I1$ and $I2$ can be violated. Here, we show that RoBcast protocol is self-stabilizing, that is, starting from any arbitrary state, after

the faults stop occurring (i.e., no faults occur for a period sufficient enough for stabilization) RoBcast starts satisfying its specification.

We prove Theorem 4.5 by proving that starting from any arbitrary state RoBcast converges to its initial state in finite time where $I1$ and $I2$ are satisfied. More specifically, $I1$ and $I2$ are re-established within at most $max\_message\_length$ rounds, where $max\_message\_length$ denotes the maximum number of packets that a message can span. Note that once the invariant $I1$ and $I2$ is satisfied, Theorem 4.3 states that the hidden-node problem is eliminated in the *DATA* phase of the subsequent rounds.

Theorem 4.5. *(***Self stabilization***) RoBcast is self-stabilizing.*

Proof. Our proof is by demonstrating a variant function g that always decreases outside the invariant states.

$$g: <number\ of\ transmitters\ within\ single\text{-}hop\ of\ a\ node\ k>$$

We show below that g always decreases until a state where $g = \langle 0 \rangle$). We first show that g cannot increase by considering all possible transitions of a node v_i that attempts to transmit *DATA*. If v_i is in the $idle$ state in the RTS phase, then it can remain so or move to $candidate$ or $veto$ states. If v_i is in the $idle$ states in the $NCTS$ phase, then it remains so at the beginning of the *DATA* phase of that round. If $v_i.veto$ holds during the $NCTS$ phase, then the node after transmitting a $NCTS$ message transits to $idle$ state when the *DATA* phase begins. If $v_i.candidate$ holds during the $NCTS$ phase, and $g \neq \langle 0 \rangle$, then depending on whether any of the nodes in $Nbr(v_i)$ were in $veto$ state, v_i can transit to either $transmit$ or $idle$ state at the beginning of the *DATA* phase of that round. But, $g \neq \langle 0 \rangle$ implies the existence of a node in the neighborhood that received a collision of RTS messages and hence in $veto$ state during the $NCTS$ phase. Thus $v_i.candidate$ is blocked from moving to

transmit state and becomes *idle* during the *DATA* phase. The situation where all the nodes in the neighborhood are transmitting an *RTS* at the same instant is avoided by having each node carrier sense, as specified in the radio model 3.2, during the *RTS* phase and back off before transmitting an *RTS*.

We now show that g decreases. Due to the upper bound on the *max_message_length*, a node can remain in *transmit* state only for a finite number of rounds. Hence, the *remaining_length_of_message* always decreases after each round. Therefore, within at most *max_message_length* rounds, g reduces to $\langle 0 \rangle$, which is the initial state where both $I1$ and $I2$ are satisfied. □

4.4 Extensions

Here, we discuss some extensions that could be applied to the RoBcast protocol.

When a node is listening to the channel, it spends as much energy as transmitting [6]. Therefore, it is important to reduce any idle listening in our MAC protocol. The primary advantage of a TDMA based protocol in WSNs is the energy efficiency due to lack of collisions and the possibility to put nodes on a synchronized sleep schedule. Though RoBcast is not TDMA based, we can reap the benefits of round-synchronization. To this end, we can extend RoBcast so that when a node, v_i, detects that it is not receiving any message transmission in the beginning of a DATA phase, it can turn off its radio, set a timer, and sleep for the rest of the *DATA* phase. Later, upon expiration of its timer, v_i can wake up at the beginning of the *RTS* phase. Similarly, when a contending node v_i is deferred from access to the channel via Action 4, v_i can turn off its radio and sleep until the beginning of the next round's *RTS* phase. This allows the node to sleep during the *DATA* phase - up to

90% of the round duration. This kind of energy savings is highly attractive for battery operated wireless sensor nodes. In future work, using PowerTOSSIM [38], we will quantify the energy-savings we achieve by eliminating idle-listening as mentioned.

Secondly, it is possible to extend RoBcast to also support unicast messages with a simple extension as in [33]. To achieve this, we introduce a third type of packet - CTS message that can be transmitted during the $NCTS$ phase. In case of a successful receipt of a unicast message, a node replies with a CTS message if it is the intended receiver. Any possible interference caused by the transmitter in its neighborhood, by this unicast transmission will be avoided by an $NCTS$ response from its neighbors. Thus, the receipt of a CTS at the transmitter gives the go ahead and the unicast message will be transmitted. If the transmitter does not receive an CTS message or receives a collision, it backs off due to the possibility of corrupting parallel transmissions. This scheme will allow the RoBcast protocol to reliably deliver both unicast and broadcast messages.

Chapter 5

Simulations

In Chapter 4, we presented the RoBcast protocol. With the aim of studying the feasibility of implementing RoBcast and comparing its performance with other protocols, we simulated the protocol in Prowler [22]. In this chapter, we present the results of the simulation. The code used for the simulation is attached in Appendix A.

5.1 Preliminaries

Simulation setup

As stated before, the simulations were carried out in Prowler [22] with the network laid out as a 5-by-5 grid of nodes, a total of 25 nodes. The traffic load was varied by varying the number of nodes requesting to transmit data.

Simulations were conducted using both ideal and realistic Mica radio. In the former,

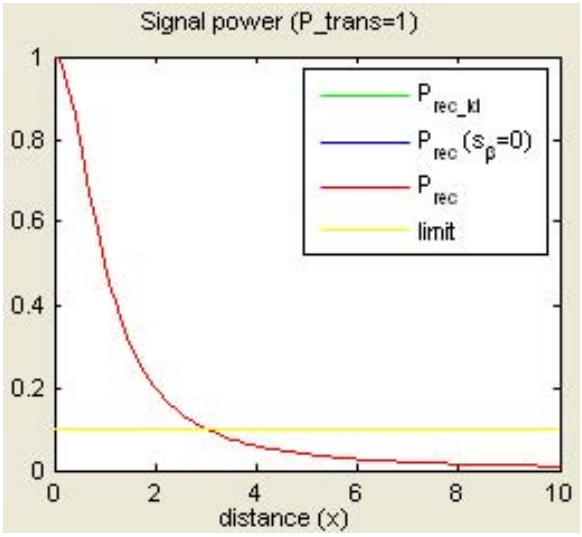

Figure 5.1: Ideal radio's characteristics. source[22].

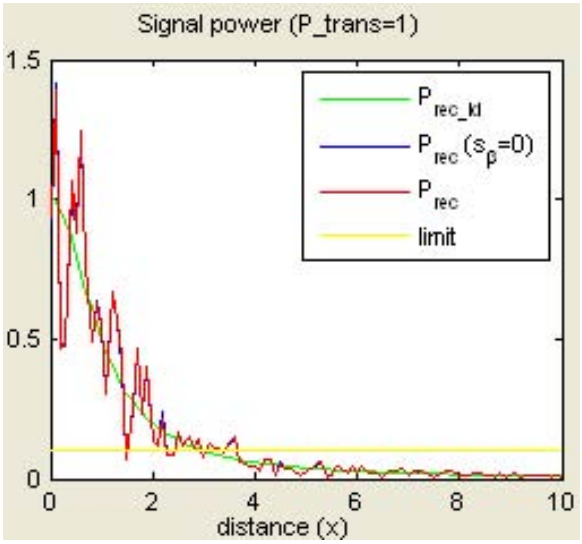

Figure 5.2: Realistic radio's characteristics. source[22].

transmissions are maintained free from external influences like noise as well as from multi-path fading. A message in this environment can be lost only when it collides with another message. However, in the realistic radio model, transmissions are subject to Rician fading and multipath interference effects as well as collisions along with a 5% error probability for each message reception.

The reported data for our experiments are values averaged over 10 independent runs for each configuration.

Metrics

In our simulations we compare the performance of the various protocols over a period of time. However, presenting the data collected against the simulation time does not reflect the statistic accurately. Hence, we attempt to measure the time during which the network is active with "*settling time*" which is defined as the duration between the last time a packet

is received and the first time a packet is sent. We also define *goodput* as the rate at which data bits are received by the nodes.

With these definitions in mind, we have compared the following metrics for the various protocols.

- **Throughput :** This provides us an idea of the channel's bandwidth usage and is calculated as *total number of bits received/settling time*. .
- **Goodput :** Calculated as *number of data bits received/settling time*, goodput presents the effective usage of the bandwidth for data bits, ignoring the effect of control bits.
- **Message Latency :** The time taken by a node with a message to transmit to actually propagate that message to its intended recipients is represented as the message latency and is calculated as the delay in transmitting DATA since its first attempt to transmit the data.
- **Control Overhead :** As the name suggests, this metric, calculated as *number of control bits transmitted/data bit received* reflects the number of control bits required for the successful transmission of a data bit.
- **Total Loss of Packets :** In our attempt to understand the performance of the protocol we measure the number of data bits lost in the transmission as the number of unique data packets received for each data packet transmitted. This metric is however different from the number of collisions reported during a particular DATA phase, because multiple receivers could report the collision of the same data packet. Here, we are talking in terms of the transmitter, i.e., if a node transmits data, what is the probability that none of the intended recipients receive the data. Thus, this metric relates well to wireless sensor networks with heavy energy constraints where transmission and

reception of the longer data messages translates to the effective energy consumption of the radio.

Table 5.1 presents the message format of the protocols compared. The data payload in all the protocols is 960 bits long. In BEMA the CONTROL phase is for 100 bit-time. The RTS/CTS/NCTS and all other control messages in RoBcast, BSMA and BMMM are 48 bits long, as implemented in SMAC [8]. It is to be noted that there are no control messages in CSMA.

	Control packets	Data packets
CSMA	0 bits	960 bits
BSMA	48 bits	960 bits
BMMM	48 bits	960 bits
BEMA	100 bits	960 bits
RoBcast	48 bits	960 bits

Table 5.1: Message formats of the protocols.

5.2 Simulation Results

Throughput & Goodput

Figures 5.3 and 5.4 show the throughput for the protocols, while Figures 5.5 and 5.6 show the goodput for the protocols under ideal and realistic radio, respectively. The smaller size of the control packets, does not affect the effective throughput much. Hence the graphs for goodput seem to be shifted down for the various protocols when compared to throughput, with CSMA alone showing no change (due to the absence of control of packets).

CSMA does not attempt to provide any reliability, and transmits any data as soon as it can without eliminating the hidden node problem. Hence, it offers the highest goodput

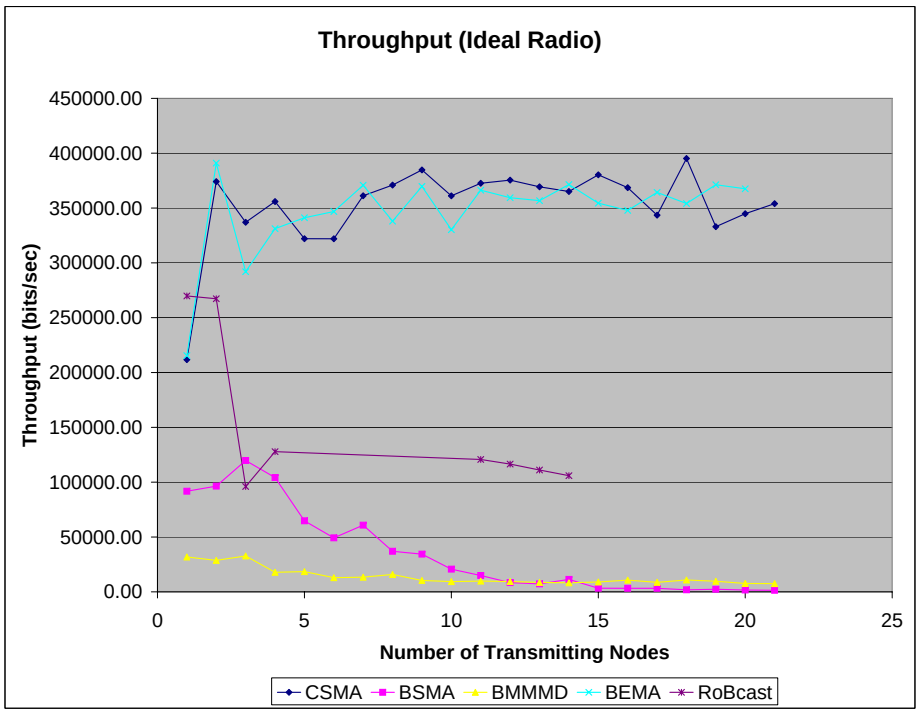

Figure 5.3: Throughput in ideal radio model.

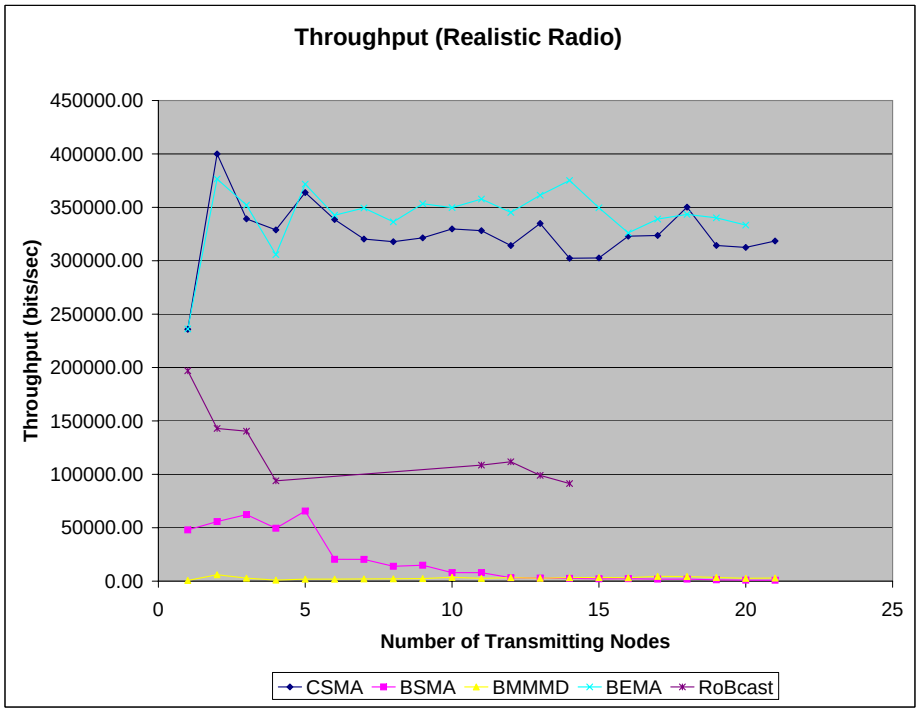

Figure 5.4: Throughput in realistic radio model.

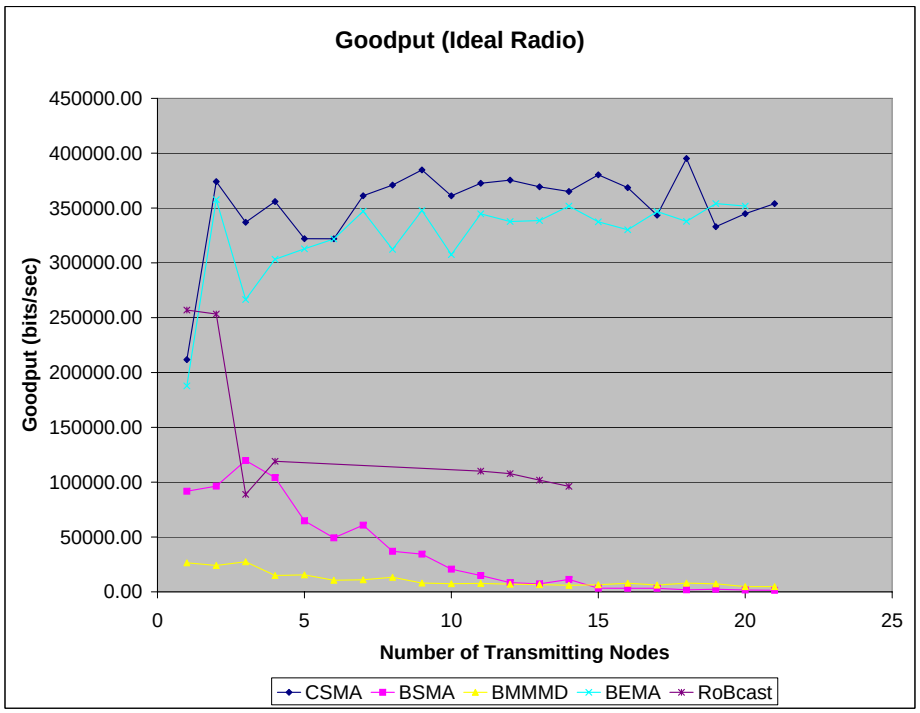

Figure 5.5: Goodput in ideal radio model.

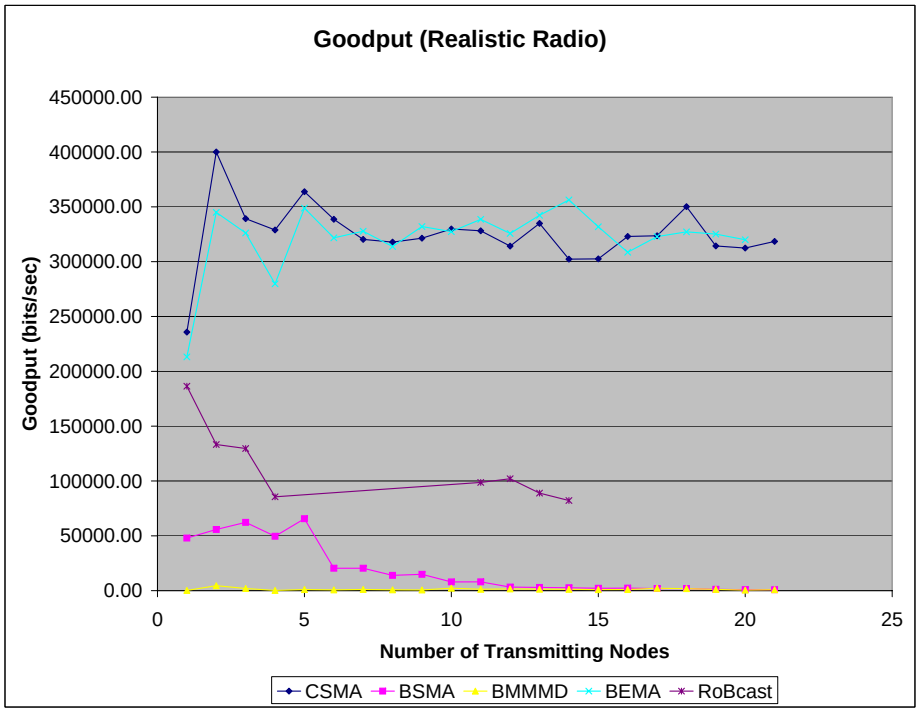

Figure 5.6: Goodput in realistic radio model.

while suffering in reliability as seen in Figures 5.11 and 5.12. The goodput of BSMA linearly decreases with respect to the number of transmitters, due to the corresponding linear increase in the number of collisions. BMMM' guarantees reliable delivery of data to all neighbors (it ensures virtually no collisions), however, due to the individual handshakes, a large synchronization overhead and latency is incurred which results in the low goodput. BEMA has a high goodput among the protocols. BEMA attempts to elect leaders during each of its rounds to reliably deliver data while decreasing collisions. It also scales well with a increase in the number of transmitters. RoBcast however, pays the price for back-offs in a round based system and has a high settling time - hence the lower goodput. Though BEMA and RoBcast are round based, BEMA elects a leader always, hence providing better goodput.

Message Latency

Figures 5.7 and 5.8 show the message latency for the protocols under ideal and realistic radio, respectively. The results are similar in both graphs with BMMM' having the highest latency due to its individual handshake with each node. BSMA's latency is the next highest. This is because BSMA has to keep retransmitting the DATA until all its neighbors receive the DATA, while the latency keeps on increasing since the first transmitted RTS. RoBcast's latency is quite low because the absence of collisions allows the data transmissions to go through as soon as the network is idle. BEMA attains lower latency than RoBcast because it elects an leader for every round while in RoBcast the back-off's could schedule rounds with no transmission in the neighborhood.

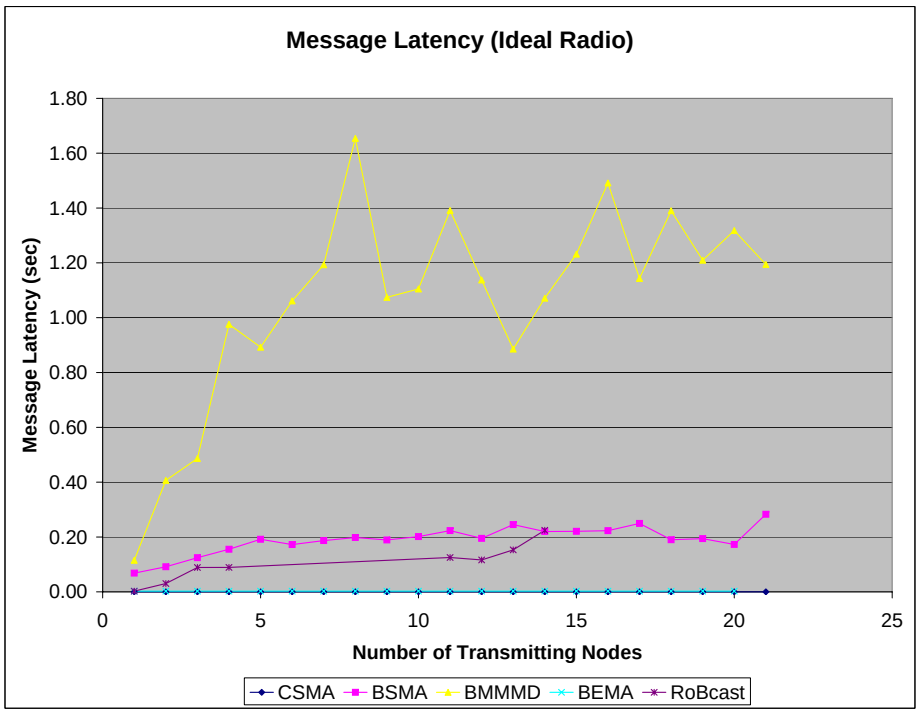

Figure 5.7: Message Latency in ideal radio model.

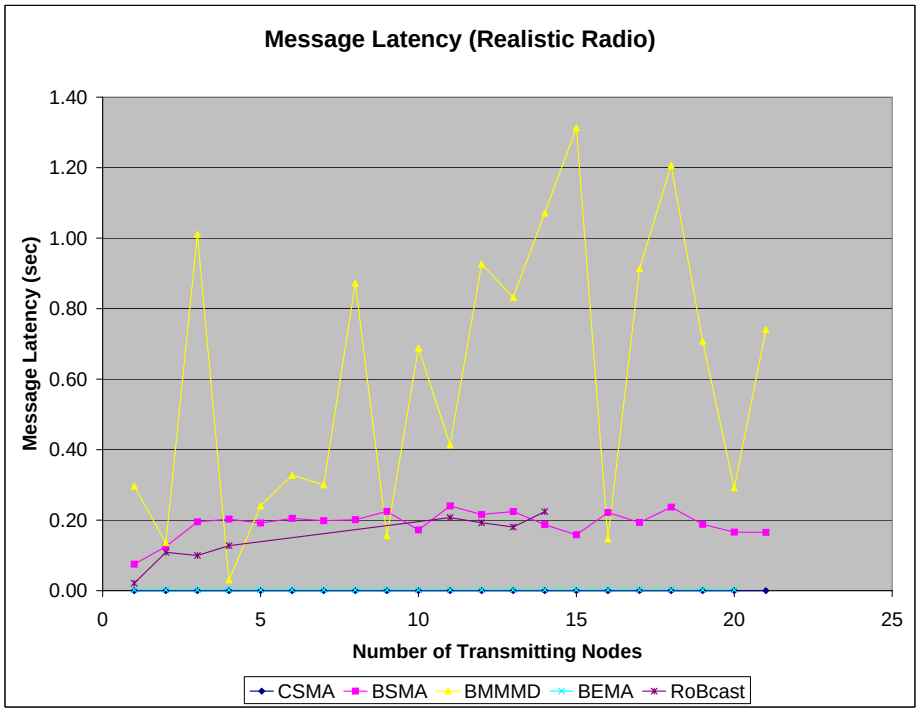

Figure 5.8: Message Latency in realistic radio model.

Control Overhead

Figures 5.9 and 5.10 show the control overhead for the protocols under ideal and realistic radio, respectively. The results are similar in both graphs with increased overhead under realistic radio, due to retransmissions. Since CSMA does not use control packets, its overhead is zero at all times. BSMA and BMMM' however, pay the penalty by using a large number of control packets to detect the possibility of collisions, before transmitting a data packet. In comparison to these protocols, RoBcast reaps the advantage of the round synchronization and maintains a lower control overhead.

Total loss of packets

Figures 5.11 and 5.12 show the number of packets completely lost after transmission for the protocols under ideal and realistic radio, respectively. The results are similar in both graphs with increased collisions under realistic radio, possibly due to nondeterministic interference among nodes and transmission error probability. Since CSMA employs no special control messages to prevent collisions, the number of collisions is highest for CSMA due to hidden node problem and the reliability linearly decreases with respect to the number of transmitters. BSMA's reliability hovers around 0.80 when the number of transmitters in the network increases beyond 40% of the network size, i.e. 10 transmitters. BMMM' shows excellent reliability under ideal conditions as it requests individual acknowledgments from the receivers to guarantee delivery. However, we notice a performance dip with realistic radio. This happens in BMMM' because, a transmitter that transmits data to its neighbor after a RTS-CTS handshake requests an acknowledgment. However, if the request for acknowledgment (RAK) or acknowledgment (ACK) is corrupted due to the realistic radio, the bits transmitted during the entire handshake, data transmission is considered a lost effort.

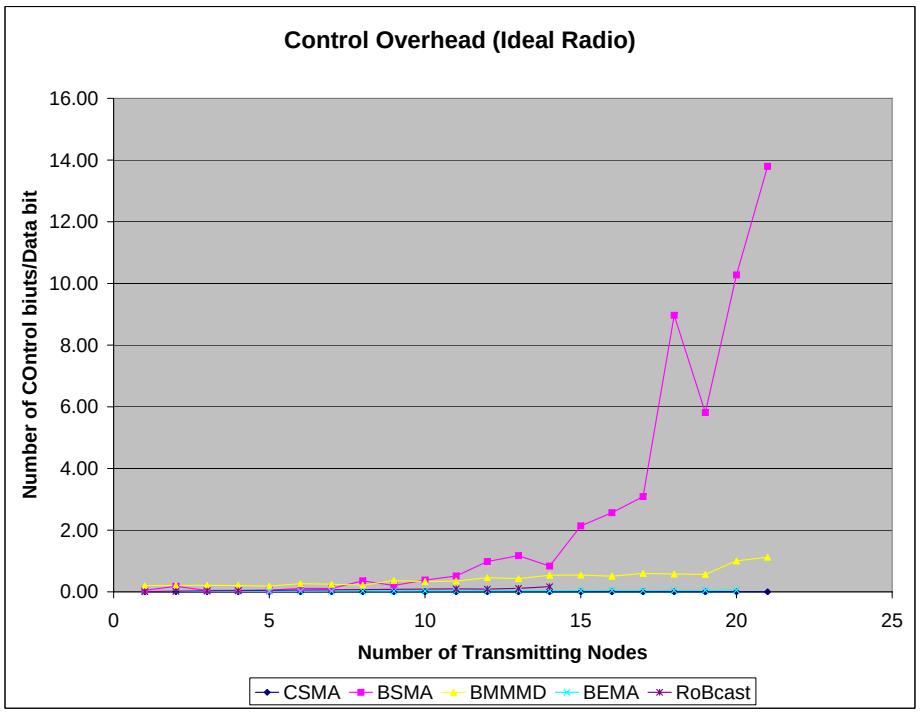

Figure 5.9: Control Overhead in ideal radio model.

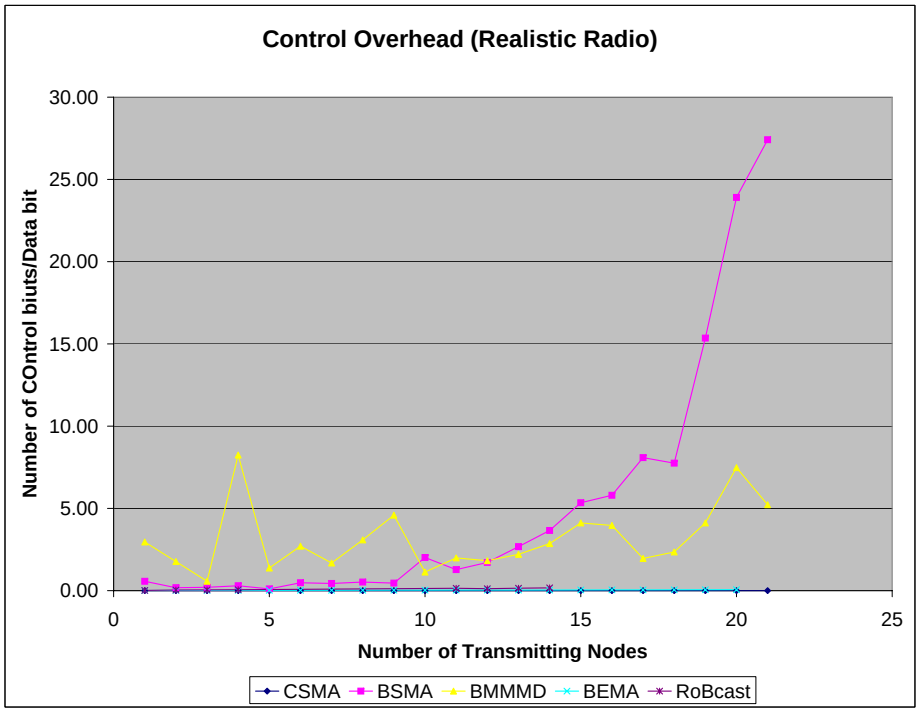

Figure 5.10: Control Overhead in realistic radio model.

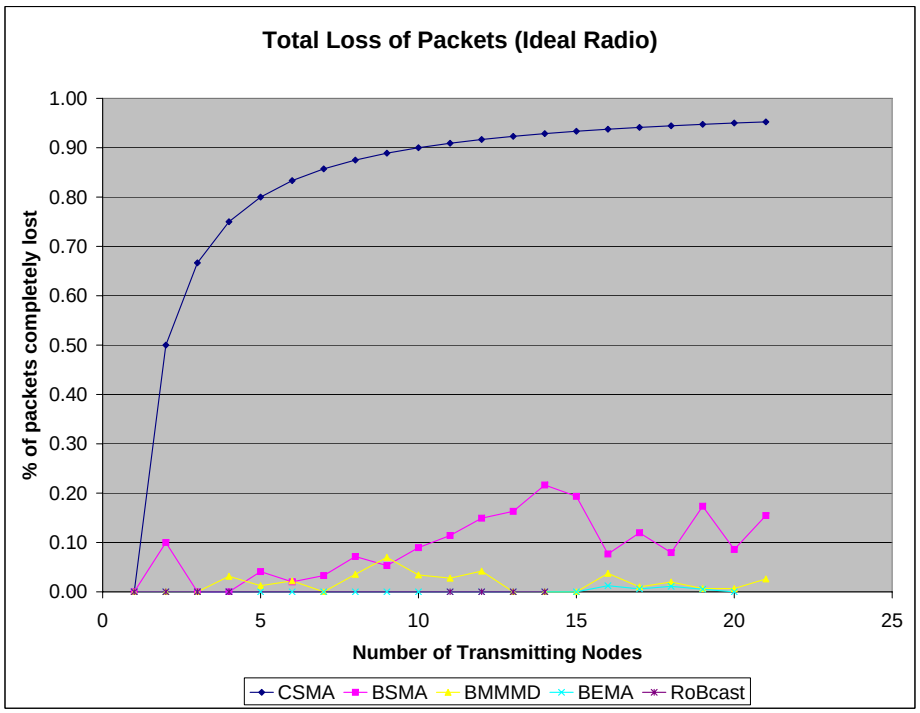

Figure 5.11: Total loss of packets in ideal radio model.

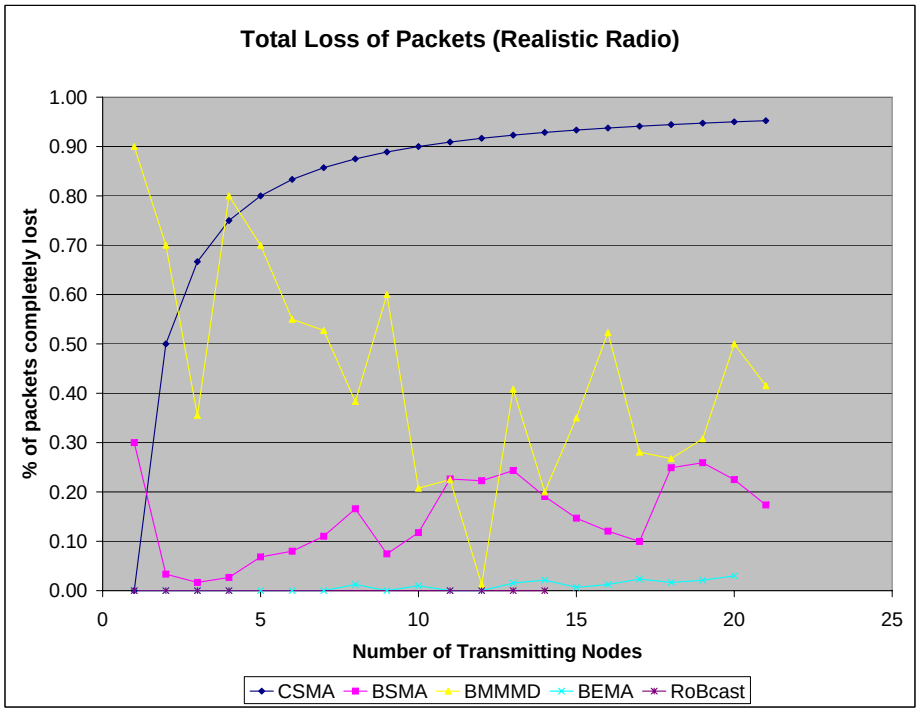

Figure 5.12: Total loss of packets in realistic radio model.

Hence, the low reliability. BEMA shows very few data collisions and offers high reliability, which is by and large constant with respect to the number of transmitters. The data collisions that arise could be due to selection of the same contention length or *unidirectionality* in some links or non-deterministic interference among nodes. RoBcast, however, offers the best reliability in comparison. This is because, RoBcast transmits a data packet only if there is no contention in the channel for that round.

Radio's power consumption

Figures 5.13 and 5.14 show the power consumption of the radio under ideal and realistic radio, respectively. RoBcast reduces the number of collisions and hence reduces the number of repeated transmissions leading to fewer messages transmitted and received as compared to the other protocols. Also, the power consumption of BEMA is the highest due to the large number of BUSY packets required during the CONTROL phase. It must, however, be noted that though the goodput of BEMA is higher than RoBcast, its power consumption is much more than RoBcast.

5.3 Multi-Packet Transmissions

The simulations and results presented indicate that RoBcast is a highly reliable protocol that eliminates collisions and transmits data reliably with a low control overhead and message latency. However, in many applications there is a need to transmit data packets longer than 960 bits. Protocols like CSMA, BSMA, BMMM' can take advantage of their non-TDMA based system and change their packet size to transmit, while BEMA and RoBcast transmit multiple packets as part of a single message. One would expect this to induce a overhead in

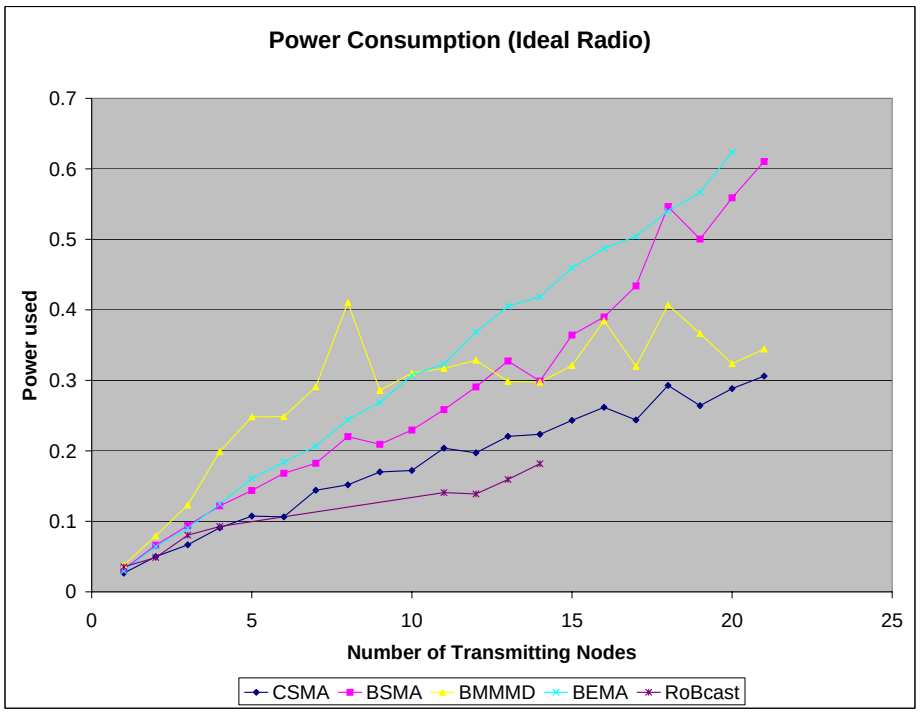

Figure 5.13: Radio's power consumption in ideal radio model.

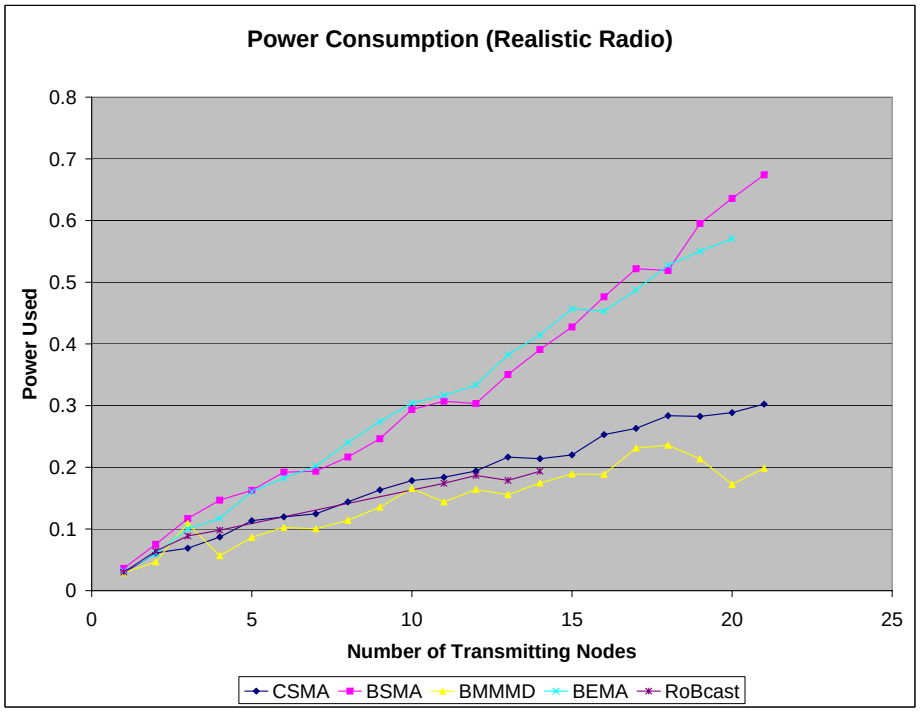

Figure 5.14: Radio's power consumption in realistic radio model.

terms of latency and number of control bits transmitted per data bit. To look at these cases further, we simulated the instance where nodes broadcast data of $960 * 4 = 3860$ bit length.

	Control packets	Data packets	Number of Parts
CSMA	0 bits	4*960 bits	1
BSMA	48 bits	4*960 bits	1
BMMM	48 bits	4*960 bits	1
BEMA	100 bits	960 bits	4
RoBcast	48 bits	960 bits	4

Table 5.2: Message formats of the protocols for Multi-Packet Transmission.

The results show that RoBcast's behavior in comparison to the other protocols is similar to the single packet transmission case. The graphs are presented below in Figures 5.15 - 5.26

5.4 Discussion

When compared to Busy Elimination Multiple Access (BEMA), RoBcast should perform better as it eliminates the hidden node problem completely. RoBcast is not susceptible to the obstruction problem as depicted in Figure 5.27, a variant of the hidden node problem, as it is based on a receiver side collision detection mechanism.

Also unlike protocols like [35], RoBcast does not transmit data if the virtual carrier sensing fails, i.e. it detects collision. So the transmission semantics is "**All** or **none**" like LBP [39] and BMMM [17]. This results in the loss of only the control packets and not data packets, hence resulting in a better goodput.

However RoBcast will incur the overhead of reserving the channel over the entire neighborhood of the transmitter unless its a broadcast transmission. This will result in poor

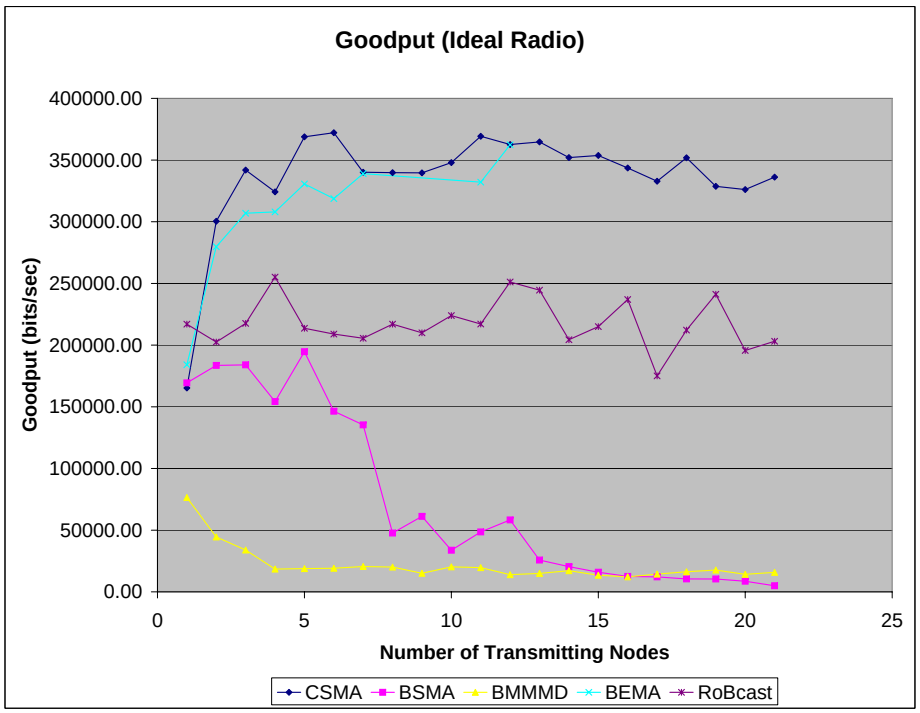

Figure 5.15: Goodput in ideal radio model for multi-packet message.

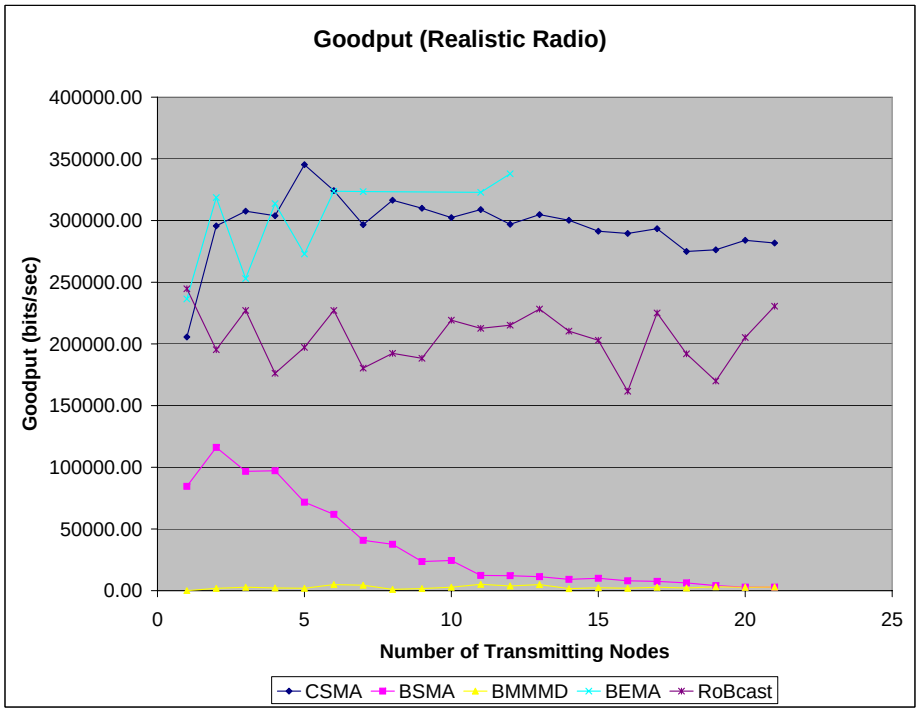

Figure 5.16: Goodput in realistic radio model for multi-packet message.

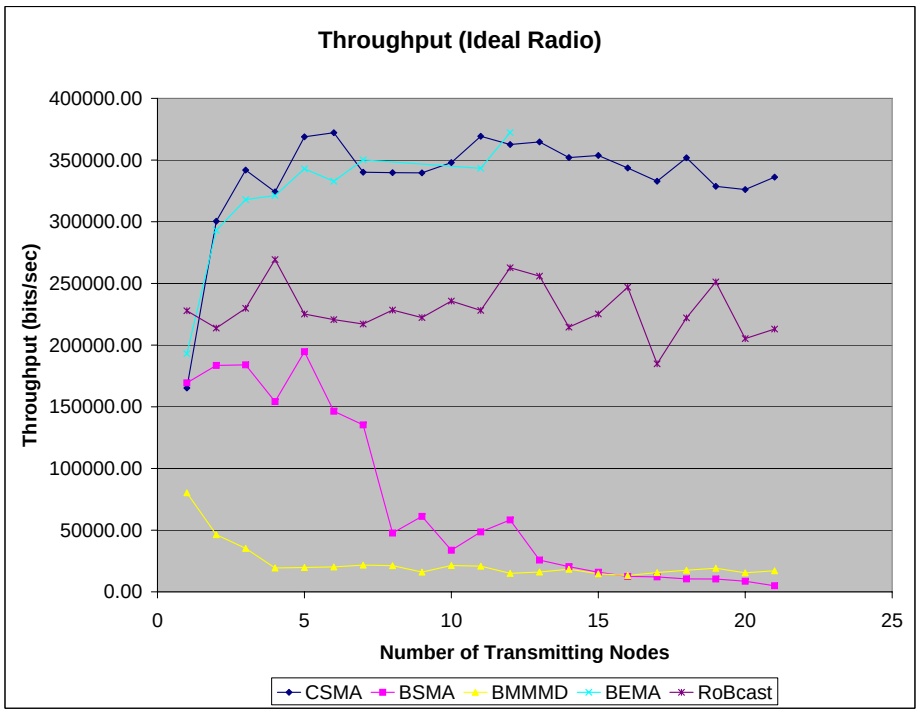

Figure 5.17: Throughput in ideal radio model for multi-packet message.

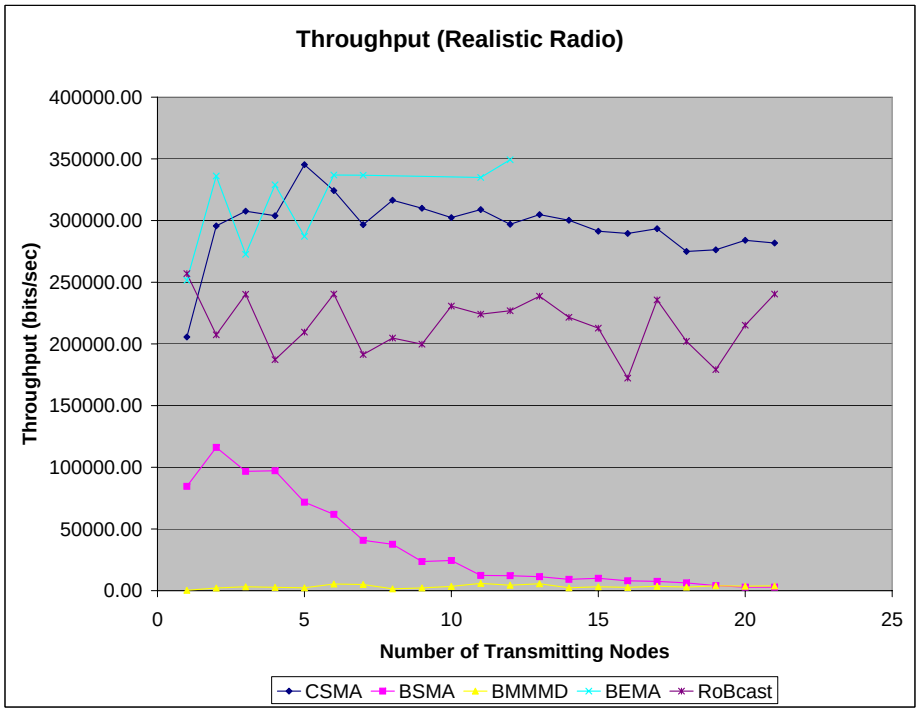

Figure 5.18: Throughput in realistic radio model for multi-packet message.

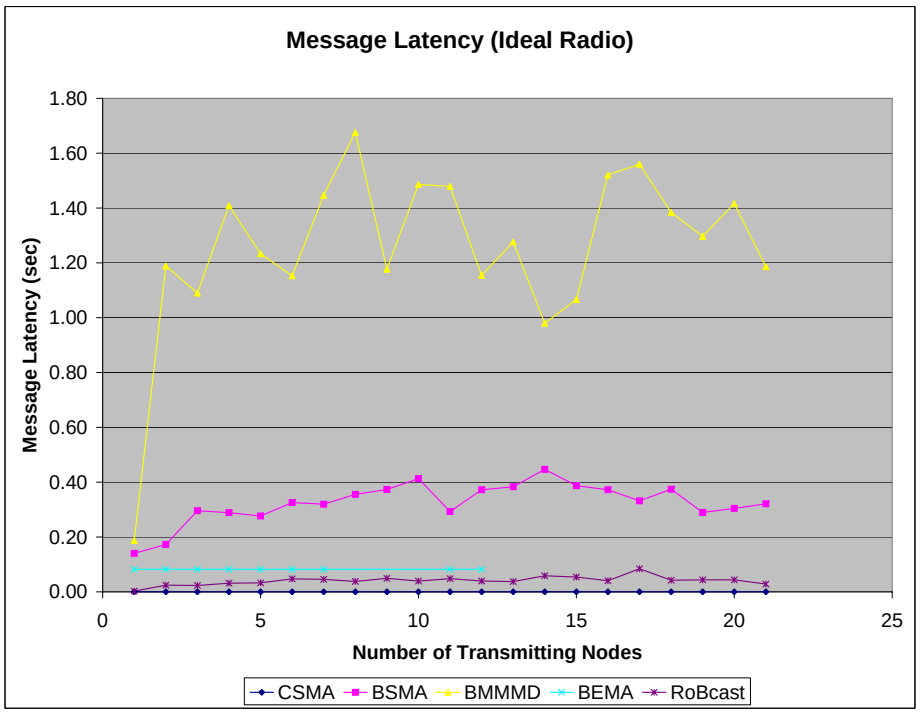

Figure 5.19: Message Latency in ideal radio model for multi-packet message.

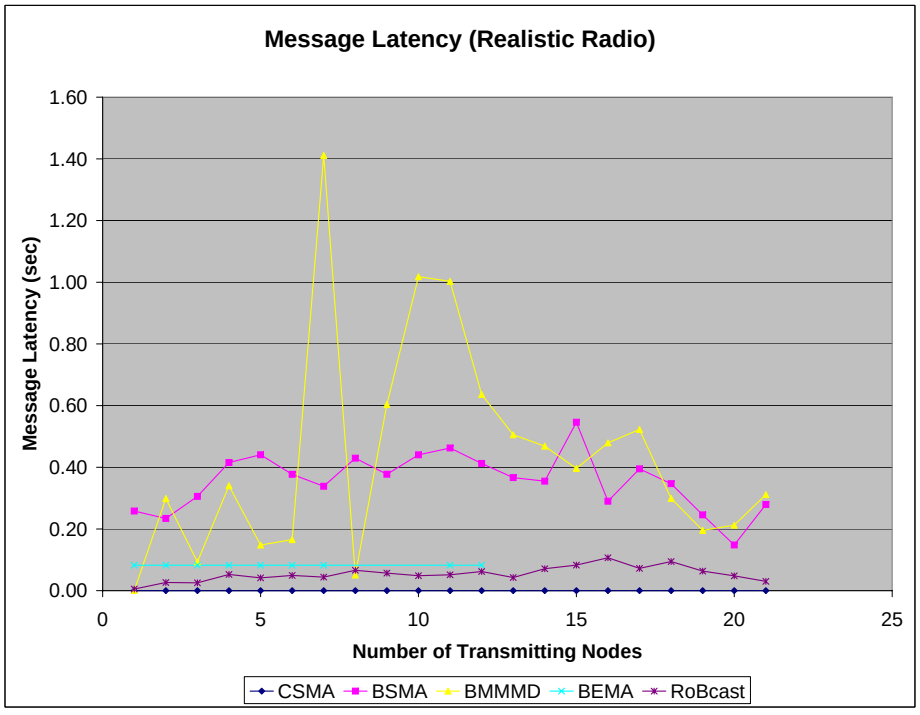

Figure 5.20: Message Latency in realistic radio model for multi-packet message.

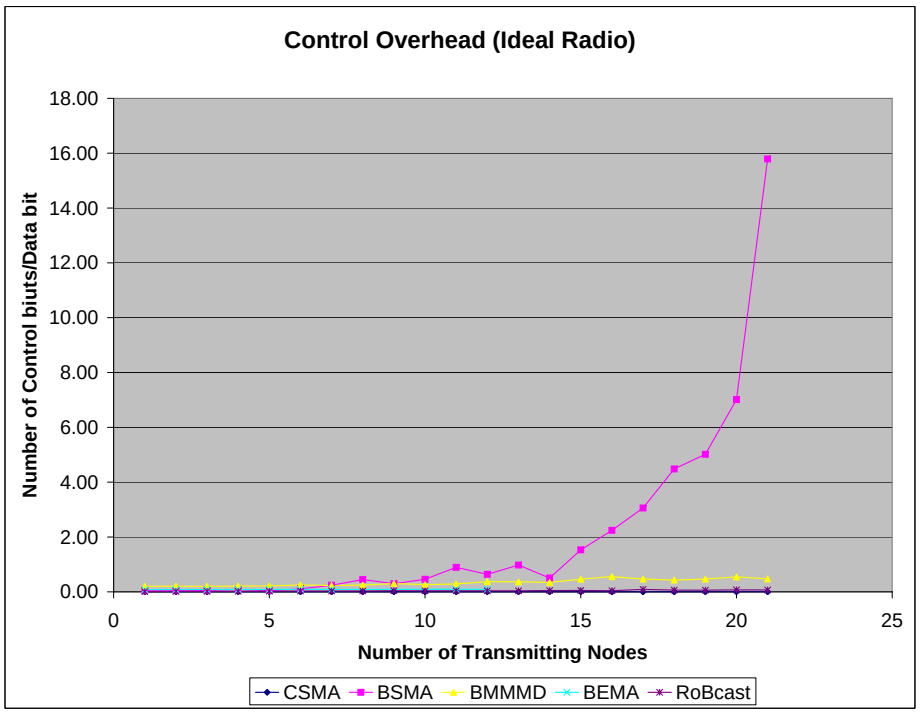

Figure 5.21: Control overhead in ideal radio model for multi-packet message.

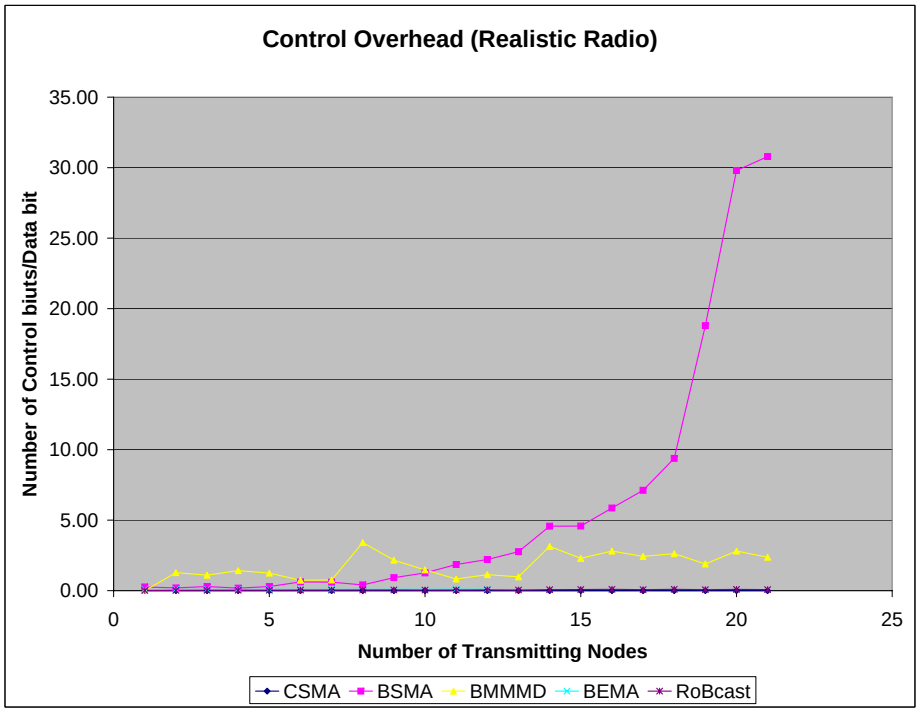

Figure 5.22: Control overhead in realistic radio model for multi-packet message.

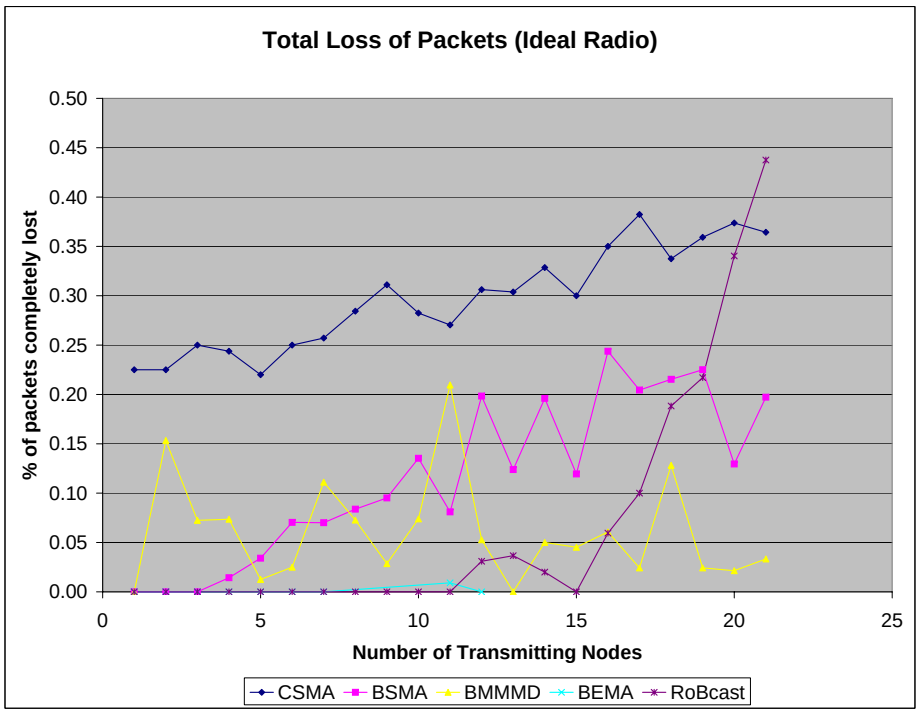

Figure 5.23: Total loss of packets in ideal radio model for multi-packet message.

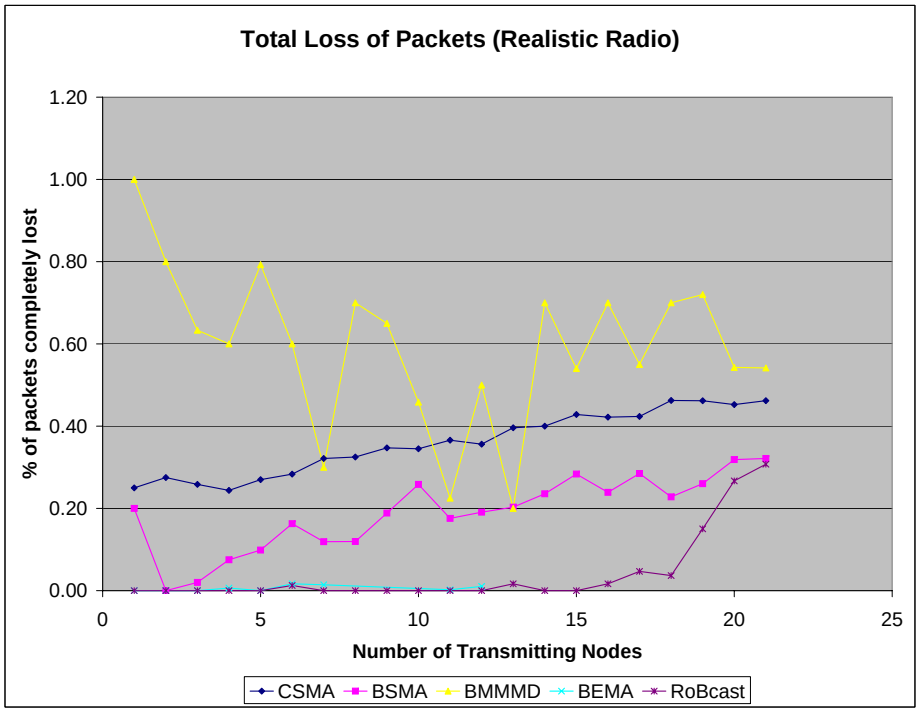

Figure 5.24: Total loss of packets in realistic radio model for multi-packet message.

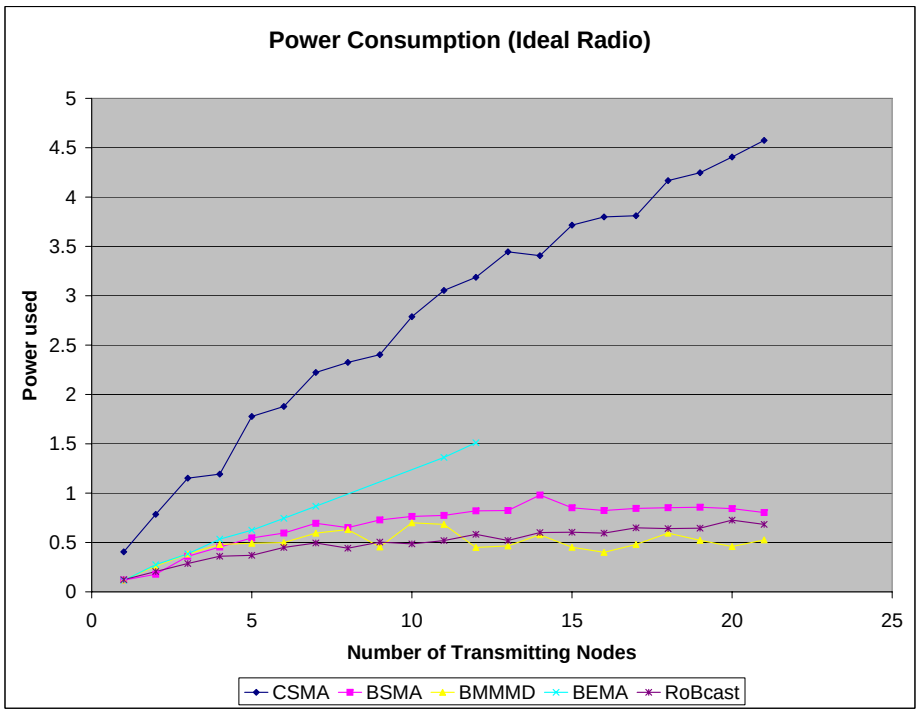

Figure 5.25: Power Consumption in ideal radio model for multi-packet message.

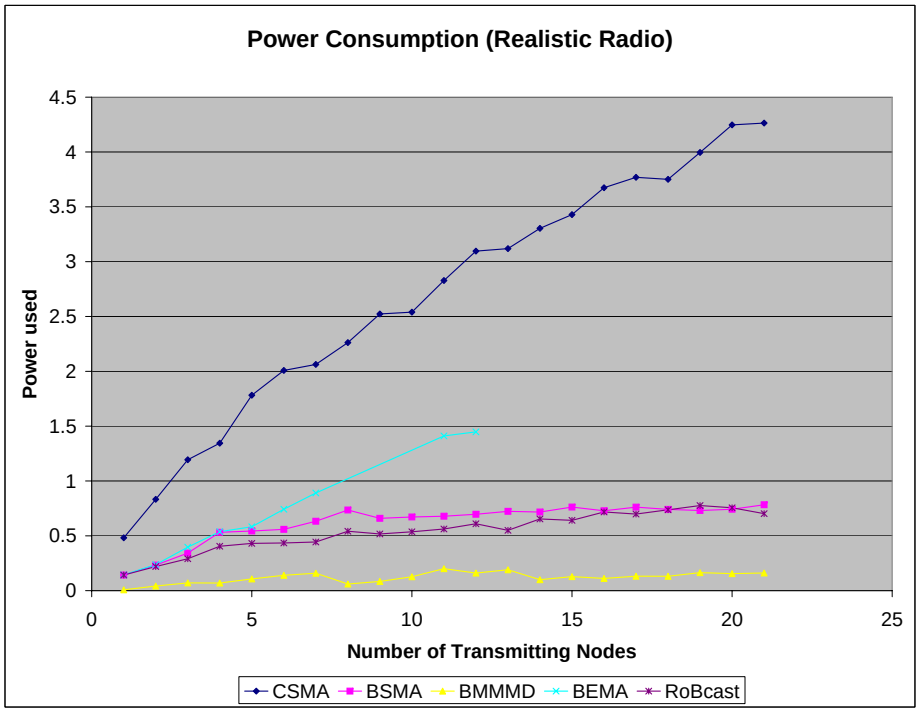

Figure 5.26: Power Consumption in realistic radio model for multi-packet message.

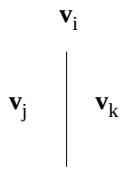

Figure 5.27: Obstacle arrangement where RoBcast solves the hidden node problem.

efficiency if a lot of unicast messages are transmitted - a price for reliability.

Chapter 6

Experiments

In this chapter we describe the various experiments conducted using mica2 [34] and tmote-sky [40] in TinyOS [41]. The experiments were performed as a initial proof-of-concept for the implementation of RoBcast.

6.1 Synchronization

As discussed in Chapter 4, RoBcast requires a synchronization mechanism that allows all the nodes in the neighborhood to have a concept of rounds. To look at the usability of a time synchronization algorithm, we compiled code to test FTSP [20] for mica2 motes and synchronized 5 nodes. We noticed that FTSP has a initialization delay in the order of a few minutes, after which the clocks remain synchronized. The maximum skew error in global time for ftsp is reported in [20] to be below 67μs. The fine level of synchronization as provided by FTSP is sufficient for RoBcast's round requirements.

6.2 Collision Detection

An important transition in RoBcast in the *NCTS* phase is based on the protocol detecting collision of messages. Collisions, as stated before, can be differentiated from channel noise by the channel energy. In view of this, we experimented with mica2 and tmote-sky motes and implemented collision detectors.

6.2.1 Collision Detectors

Based on our classification of a collision on any loss of data, we implemented collision detectors to raise a collision event if

- we receive a corrupt message as detected by CRC failure or bad packet length, or,
- we detect channel activity when the radio is not expected to receive any message.

6.2.2 Setup

For these experiments, we look at a single-hop network, and used a beacon mote to notify neighboring motes the round boundary by means of a reference broadcast similar to [21]. On receipt of the SYNC message, the motes synchronize their clocks for the round. Apart from the beacon, we used one mote to detect collisions, if any, caused by two transmitters. The setup is shown in Figure 6.1.

We configured B-MAC [6] in TinyOS for our experiments and disabling B-MAC's CCA and acknowledgments. Since we wanted to control the exact transmission times, we reset both the back-off timers - initial and congestion.

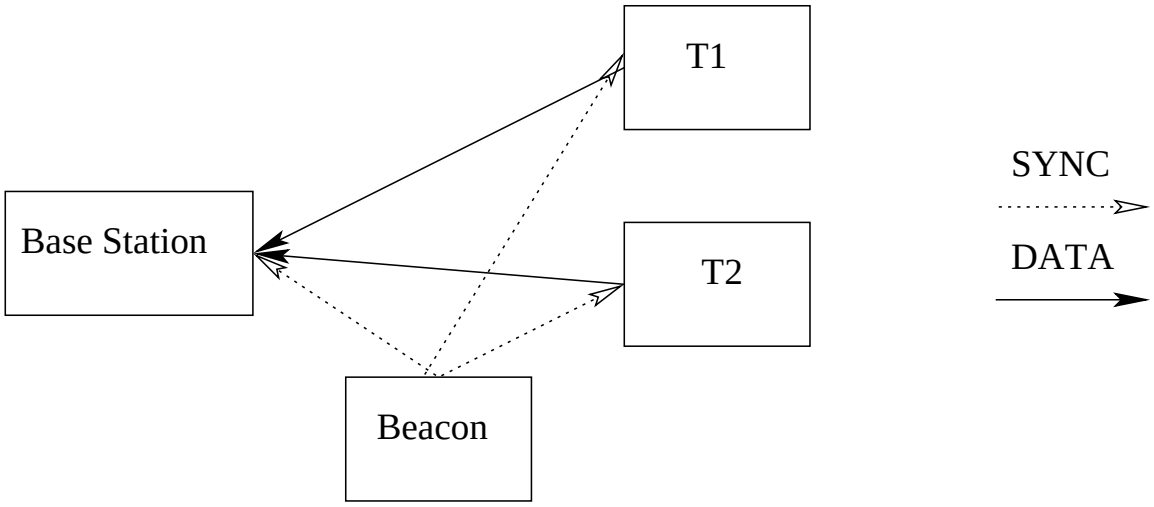

Figure 6.1: Experiment setup

6.2.3 Experiments on Mica2

On the mica2 motes, strength of a received signal is obtained from the radio and stored in the RSSI (received signal strength indicator) register. In our experiments, we polled the register periodically for the RSSI values to detect channel activity. We looked at the RSSI values to differentiate channel noise and activity in the channel. Channel noise captured is shown in the figure 6.2

As seen, the channel noise is seen to be between -90 and -100 dBm. However, we also noticed a increase in the channel energy of noise when other radio's are turned on in the neighborhood as shown in figure 6.3

Apart from such variations in noise levels, a realistic radio's varying power of transmission can cause problems while detecting collisions. We assume that the transmission power is maintained constant and attempted to detect collisions based on the detectors described. In our proof-of-concept implementation, based on the B-MAC's CCA, we detected channel activity when there are transmissions in the channel. However, when the capture effect [42]

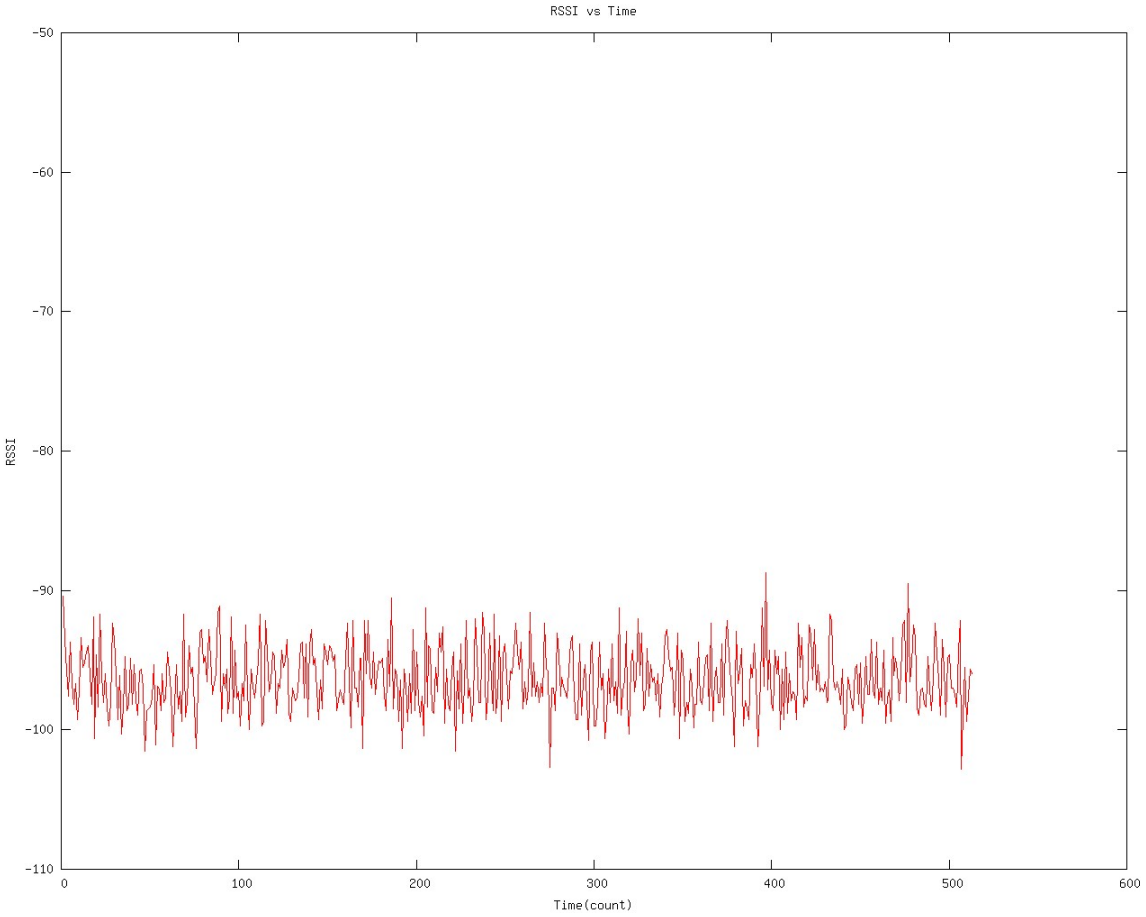

Figure 6.2: Channel energy of noise

is present, we are unable to detect the loss of the weaker signal.

6.2.4 Conclusion of CD Experiments

From the current and past experiments we infer that while it is possible to detect most of the collisions based on the RSSI of the signal using the CC1000, further experiments need to be performed on other radios like CC2420.

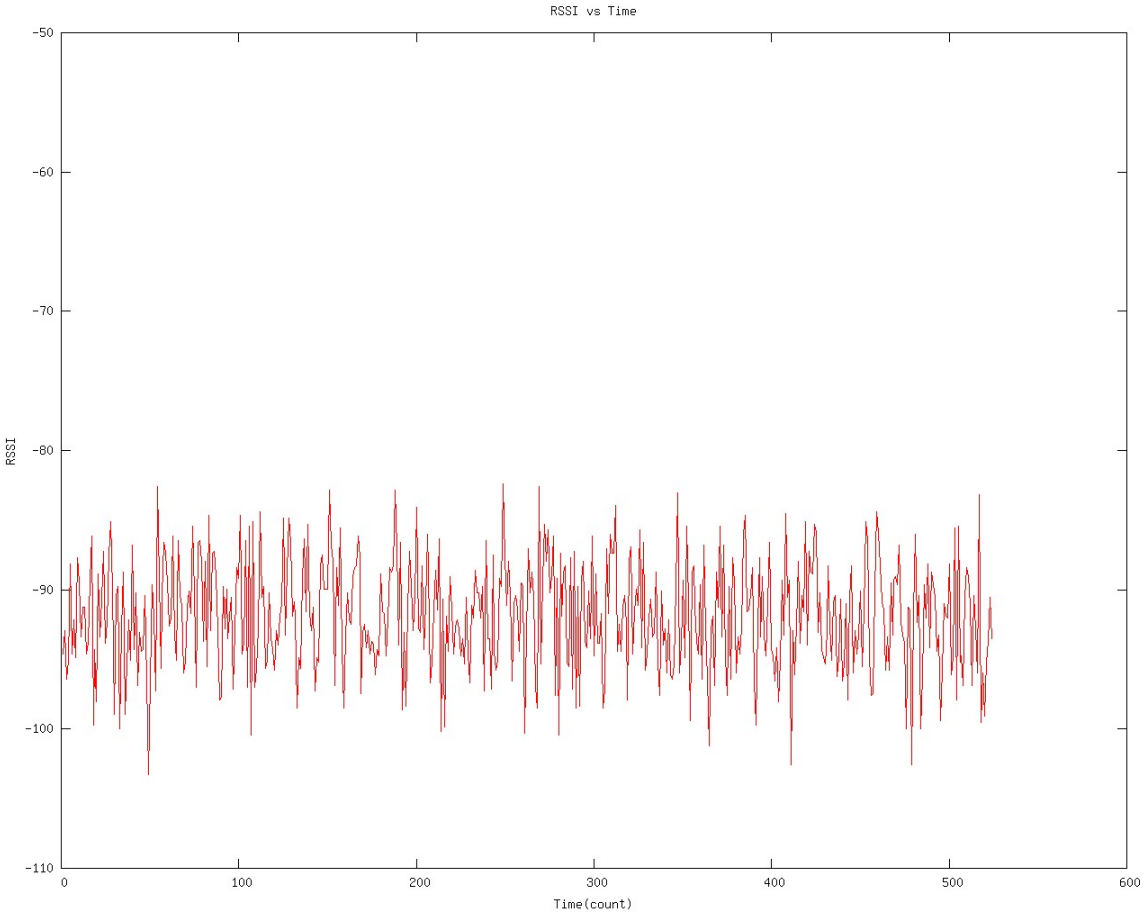

Figure 6.3: Increased channel energy of noise

Chapter 7

Conclusion & Future Work

We presented a self-stabilizing MAC protocol, Reliable Broadcast (RoBcast) protocol, that solves the reliable broadcast problem. RoBcast provides on-demand access to the channel using a RTS-NCTS handshake while taking advantage of the synchronous nature of the round based system. By avoiding collisions during the DATA phase and eliminating the hidden-terminal problem, RoBcast provides a useful building block for applications with reliability requirements. Simulations show that RoBcast's overhead is small and has the least total packet loss among BEMA [43], BSMA [15], BMMM [17], and CSMA/CA [7].

In future work, we will implement RoBcast in TinyOS [41] on top of BMAC [6]. The problem of differentiating fading and collisions still exists. The system will consider message corruption due to fading - a fault and eventually stabilize. However, an elegant solution will ideally handle collisions and data lost due to fading differently. We also need to look carefully at how RoBcast can be extended to a multi-hop network. Though RoBcast will stabilize and satisfy the specifications of the reliable broadcast problem - efficiency might

be a concern. Nodes that have already received the packet might be involved in the negative feedback and degrade the performance. We will further investigate the performance improvements RoBcast provides for handling bursty traffic patterns in sensor networks. We will also work on a light-weight ad hoc synchronization scheme for implementing rounds in RoBcast.

Bibliography

[1] R. Szewczyk, A. Mainwaring, J. Polastre, and D. Culler, "An analysis of a large scale habitat monitoring application," *In Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.

[2] J. Burrell, T. Brooke, and R. Beckwith, "Vineyard computing: Sensor networks in agricultural production," *IEEE Pervasive computing*, vol. 3, pp. 38–45, 2003.

[3] G. Simon, M. Maroti, A. Ledeczi, B. Kusy, A. Nadas, G. Pap, J. Sallai, and K. Frampton, "Sensor network-based countersniper system," *Sensys*, pp. 1–12, 2004.

[4] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y.-R. Choi, T. Herman, S. S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita, "A line in the sand: A wireless sensor network for target detection, classification, and tracking," *Computer Networks (Elsevier)*, vol. 46, no. 5, pp. 605–634, 2004.

[5] A. Arora and et. al., "Exscal: Elements of an extreme scale wireless sensor network," *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2005.

[6] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pp. 95–107, 2004.

[7] "Wireless lan medium access control(mac) and physical layer (phy) specification." IEEE Std 802.11, 1999.

[8] W. Ye, J. Heidemann, and D. Estrin, "An energy-efficient mac protocol for wireless sensor networks," in *INFOCOMM*, pp. 1567–1576, 2002.

[9] D. Culler, P. Dutta, C. T. Ee, R. Fonseca, J. Hui, P. Levis, J. Polastre, S. Shenker, I. Stoica, G. Tolle, and J. Zhao, "Towards a sensor network architecture: Lowering the waistline," *The Tenth Workshop on Hot Topics in Operating Systems (HotOS X)*, 2005.

[10] S. Farritor and S. Goddard, "Intelligent highway safety markers," *IEEE Intelligent Systems*, vol. 19, no. 6, pp. 8–11, 2004.

[11] J. Zhao and R. Govindan, "Understanding packet delivery performance in dense wireless sensor networks," in *Sensys*, pp. 1–13, 2003.

[12] A. Woo, T. Tong, and D. Culler, "Taming the underlying challenges of reliable multihop routing in sensor networks," in *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pp. 14–27, ACM Press, 2003.

[13] H. Zhang, A. Arora, Y. Choi, and M. G. Gouda, "Reliable bursty convergecast in wireless sensor networks," in *MobiHoc '05: Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing*, (New York, NY, USA), pp. 266–276, ACM Press, 2005.

[14] K. Tang and M. Gerla, "Mac layer broadcast support in 802.11 wireless networks," *Proc. IEEE MILCOM*, pp. 544–548, 2000.

[15] K. Tang and M. Gerla, "Random access mac for efficient broadcast support in ad hoc networks," *Proc. IEEE WCNC*, pp. 454–459, 2000.

[16] K. Tang and M. Gerla, "Mac reliable broadcast in ad hoc networks," *Proc. IEEE MILCOM*, pp. 1008–1013, 2001.

[17] M.-T. Sun, L. Huang, A. Arora, and T.-H. Lai, "Reliable MAC layer multicast in IEEE 802.11 wireless networks," *Proc. ICPP*, pp. 527–537, 2002.

[18] F. A. Tobagi and L. Kleinrock, "Packet switching in radio channels: part II the hidden terminal problem in carrier sense multiple-access and the busy-tone solution," *IEEE trans. on commun.*, vol. COM-23, pp. 1417–1433, 1975.

[19] G. Chockler, M. Demirbas, S. Gilbert, C. Newport, and T. Nolte, "Consensus and collision detectors in wireless ad hoc networks," in *PODC*, pp. 197–206, 2005.

[20] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi, "The flooding time synchronization protocol," *SenSys*, 2004.

[21] J. Elson, L. Girod, and D. Estrin, "Fine-grained network time synchronization using reference broadcasts," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 147–163, 2002.

[22] G. Simon, P. Volgyesi, M. Maroti, and A. Ledeczi, "Simulation-based optimization of communication protocols for large-scale wireless sensor networks," *IEEE Aerospace Conference*, pp. 255–267, March 2003.

[23] M. Ali, U. Saif, A. Dunkels, T. Voigt, K. Römer, K. Langendoen, J. Polastre, and Z. A. Uzmi, "Medium access control issues in sensor networks," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 2, pp. 33–36, 2006.

[24] J. Postel, "Internet protocol DARPA internet program protocol specification," RFC 791, Internet Engineering Task Force (IETF), Sept. 1981.

[25] J. Tourrilhes, "Robust broadcast : Improving the reliability of broadcast transmissions on CSMA/CA," Tech. Rep. HPL-98-38, Hewlett Packard Laboratories, Feb. 24 1998.

[26] S. S. Kulkarni and M. Arumugam, "Ss-tdma: A self-stabilizing mac for sensor networks," in *IEEE Press. To appear*, 2005.

[27] C. Busch, M. Magdon-Ismail, F. Sivrikaya, and B. Yener, "Contention-free mac protocols for wireless sensor networks.," in *DISC*, pp. 245–259, 2004.

[28] V. Rajendran, K. Obraczka, and J. J. Garcia-Luna-Aceves, "Energy-efficient collision-free medium access control for wireless sensor networks," in *SenSys*, pp. 181–192, 2003.

[29] L. F. W. van Hoesel and P. J. M. Havinga, "A TDMA-based MAC protocol for WSNs," in *SenSys*, pp. 303–304, 2004.

[30] T. van Dam and K. Langendoen, "An adaptive energy-efficient mac protocol for wireless sensor networks," in *SenSys*, pp. 171–180, 2003.

[31] W. B. Heinzelman, A. P. Chandrakasan, and H. Balakrishnan, "Application specific protocol architecture for wireless microsensor networks," *IEEE Transactions on Wireless Networking*, 2002.

[32] I. Chlamtac, A. Myers, V. Syrotiuk, and G. Zaruba, "An adaptive medium access control (mac) protocol for reliable broadcast in wireless networks," in *IEEE International Conference on Communications*, 2000.

[33] A. D. Myers, G. V. Záruba, and V. R. Syrotiuk, "An adaptive generalized transmission protocol for ad hoc networks," *Mob. Netw. Appl.*, vol. 7, no. 6, pp. 493–502, 2002.

[34] "Crossbow technology, Mica2 platform." www.xbow.com/Products/Wireless_Sensor_Networks.htm.

[35] S. Park and R. R. Palasdeokar, "Reliable one-hop broadcasting (rob) in mobile ad hoc networks," in *PE-WASUN '05: Proceedings of the 2nd ACM international workshop on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks*, (New York, NY, USA), pp. 234–237, ACM Press, 2005.

[36] S. Ganeriwal, R. Kumar, and M. B. Srivastava, "Timing-sync protocol for sensor networks," in *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, (New York, NY, USA), pp. 138–149, ACM Press, 2003.

[37] G. Chockler, M. Demirbas, S. Gilbert, and C. Newport, "A middleware framework for robust applications in wireless ad hoc networks," in *43rd Allerton Conf. on Communication, Control, and Computing*, 2005.

[38] V. Shnayder, M. Hempstead, B. Chen, G. W. Allen, and M. Welsh, "Simulating the power consumption of large-scale sensor network applications," in *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, (New York, NY, USA), pp. 188–200, ACM Press, 2004.

[39] D. Kimt, J. Park, and Y. Choir, "An efficient reliable multicasting protocol for micro-cellular wireless networks."

[40] "moteiv, tmote sky platform." http://www.moteiv.com/products-tmotesky.php.

[41] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for network sensors," *ASPLOS*, pp. 93–104, 2000.

[42] *Exploiting the Capture Effect for Collision Detection and Recovery*, 2005.

[43] M. Demirbas and M. Hussain, "A mac layer protocol for priority-based reliable broadcast in wireless ad hoc networks," Tech. Rep. 25, SUNY Buffalo, 2005.

Appendix A

Simulation File

```
0001 function application(S)
0002 % application to simulate RoBcast - a robust broadcast protocol.
0003
0004 S; %%%%%%%%%%%%%%%%%%%%   housekeeping  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
0005 S;       persistent app_data
0006 S;       global ID t
0007 S;       [t, event, ID, data]=get_event(S);
0008 S;       [topology, mote_IDs]=prowler('GetTopologyInfo');
0009 S;       ix=find(mote_IDs==ID);
0010 S;       if ~strcmp(event, 'Init_Application')
0011 S;          try memory=app_data{ix}; catch memory=[]; end,
0012 S;       end
0013 S; %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
0014
0015 global N_CTRL_PKTS_XMIT_RBCAST;
0016 global N_DATA_PKTS_XMIT_RBCAST;
0017 global N_DATA_PKTS_RCVD_RBCAST;
0018 global N_CTRL_PKTS_RCVD_RBCAST;
0019 global N_CTRL_PKTS_COL_RBCAST;
0020 global N_DATA_PKTS_COL_RBCAST;
0021
0022 global DATA_PKTS_RCVD;
0023 global CTRL_PKTS_RCVD;
```

```
0024
0025 global MSG_ID;
0026 global ROUND_ID;
0027
0028 global MSG_LATENCY;
0029
0030 TRUE = 1;
0031 FALSE = 0;
0032
0033 MESSAGE_LENGTH = 4;%ceil(rand*10);
0034 DATA_PKT_SIZE = 960;%*4;
0035 CTRL_PKT_SIZE = 48;
0036
0037 MAX_BACKOFF_ROUNDS = 5;
0038
0039 % ROUNDS
0040 % $$RTS$$ $$NCTS$$ $$DATA$$
0041 ROUND_SIZE = 2*(CTRL_PKT_SIZE + 2) + (DATA_PKT_SIZE + 2);
0042
0043 %Type of PACKET
0044 PKT_TYPE_RTS = 0;
0045 PKT_TYPE_NCTS = 1;
0046 PKT_TYPE_DATA = 2;
0047
0048 %Type of TimeSlot
0049 RTS_TS = 0;
0050 NCTS_TS = 1;
0051 DATA_TS = 2;
0052
0053 %State
0054 IDLE = 0;
0055 CANDIDATE = 1;
0056 VETO = 2;
0057 TRANSMIT = 3;
0058
0059 global LOG_SCREEN
0060 global LOG_FILE
0061
0062 LOG_SCREEN = 1;
0063 LOG_FILE = 2;
0064
0065 global LOG_LEVEL
0066 %LOG_LEVEL = bitor(LOG_SCREEN, LOG_FILE);
```

```
0067 LOG_LEVEL = 0;
0068 %LOG_LEVEL = LOG_FILE;
0069 %LOG_LEVEL = LOG_SCREEN;
0070
0071 global fid
0072 global logfile
0073 %logfile='rbcast4_log';
0074
0075 switch event
0076 case 'Init_Application'
0077    if (LOG_LEVEL ~= 0)
0078       if (ID == 1)
0079          datetime = fix(clock);
0080          logfile = sprintf('rbcast4_log_%d%.2d%.2d_%.2d%.2d%.2d', ...
0081             datetime(1),datetime(2),datetime(3), datetime(4),...
0082             datetime(5),datetime(6));
0083       end
0084       fid = fopen(logfile, 'a');
0085       if (fid == -1)
0086          error('cannot open file for writing');
0087       end
0088    end
0089    DATA_PKTS_RCVD = [];
0090    CTRL_PKTS_RCVD = [];
0091
0092    MSG_ID = 1;
0093    ROUND_ID = 0;
0094
0095    MSG_LATENCY = [];
0096
0097    signal_strength=1;
0098
0099 %%%%%%%%%%%%%%%   Memory should be initialized here  %%%%%%%%%%%%%%%%%
0100    memory=struct( ...
0101       'currentTS',            -1, ...
0102       'excited',              FALSE, ...
0103       'state',                0, ...
0104       'backoff_rounds',       0, ...
0105       'signal_strength',      signal_strength, ...
0106       'current_time',         -1, ...
0107       'sender_id',            -1, ...
0108       'data_receive_timeout', FALSE,...
0109       'collision_detected',   FALSE,...
```

```
0110        'data_to_transmit',      0,...
0111        'data_to_receive',       0,...
0112        'rts_xmit_time',         -1,...
0113        'data_xmit_time',        -1);
0114 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
0115     memory.current_time = 0;
0116     N_CTRL_PKTS_XMIT_RBCAST = 0;
0117     N_DATA_PKTS_XMIT_RBCAST = 0;
0118     N_DATA_PKTS_RCVD_RBCAST = 0;
0119     N_CTRL_PKTS_RCVD_RBCAST = 0;
0120     N_CTRL_PKTS_COL_RBCAST = 0;
0121     N_DATA_PKTS_COL_RBCAST = 0;
0122
0123     Start_RTSTimeslot_In(0);
0124
0125     array_excited_nodes = sim_params('get_app','ENA');
0126     %array_excited_nodes = [3 24 16 20]
0127
0128     if(isempty(array_excited_nodes))
0129        error(['Error ! No nodes are excited!!!']);
0130     end
0131
0132     if(~isempty(find(array_excited_nodes == ID)))
0133        memory.excited = TRUE;
0134     end
0135
0136     %if an 'selected' transmitter
0137     %   decide if we have to transmit data in this round
0138     if (memory.excited == TRUE)
0139        memory.data_to_transmit = MESSAGE_LENGTH;
0140     end
0141
0142     sim_params('set','MAC_MIN_WAITING_TIME',0);
0143     sim_params('set','MAC_RAND_WAITING_TIME',0);
0144     sim_params('set','MAC_MIN_BACKOFF_TIME',0);
0145     sim_params('set','MAC_RAND_BACKOFF_TIME',0);
0146     %          sim_params('set','RECEPTION_LIMIT',0.22);
0147     %          sim_params('set','RADIO_SS_VAR_CONST',0);
0148     %          sim_params('set','RADIO_SS_VAR_RAND',0);
0149     %          sim_params('set','TR_ERROR_PROB',0);
0150
0151 case 'RTS_Timeslot'
0152     PrintMessage('RTS')
```

```
0153    sim_params('set','MAC_PACKET_LENGTH',CTRL_PKT_SIZE);
0154
0155    if (ID == 1)
0156       ROUND_ID = ROUND_ID + 1;
0157    end
0158
0159    %Set the current timeslot
0160    memory.currentTS = RTS_TS;
0161    % Start the Busy Timeslot at the End
0162    Start_NCTSTimeslot_In(t + CTRL_PKT_SIZE + 2);
0163
0164    if (memory.data_to_receive > 0 && ...
0165          memory.data_receive_timeout == TRUE)
0166       %Transmitter had backed off during last round and
0167       %no data was received.
0168       %so reset the data_to_receive. - Basically timeout on receive !
0169       memory.data_to_receive = 0;
0170       memory.sender_id = -1;
0171    end
0172    memory.data_receive_timeout = FALSE;
0173
0174    %Maintain the backoff counter
0175    %Decrement the counter once each round.
0176    if (memory.backoff_rounds > 0)
0177       memory.backoff_rounds = memory.backoff_rounds  - 1;
0178    end
0179
0180    % do we want to transmit data ?
0181    if (memory.data_to_transmit > 0 && ...
0182          memory.backoff_rounds == 0)
0183       % are we in the middle of a reception ?
0184       if (memory.data_to_receive == 0)
0185          PrintMessage('S RTS')
0186          N_CTRL_PKTS_XMIT_RBCAST = N_CTRL_PKTS_XMIT_RBCAST + 1;
0187
0188          %To calculate the msg_latency - log the RTS transmit time
0189          if (memory.rts_xmit_time == -1)
0190             memory.rts_xmit_time = t;
0191          end
0192
0193          Send_Packet(radiostream(struct('ID',ID, ...
0194             'type',PKT_TYPE_RTS,...
0195             'MSG_ID', MSG_ID), ...
```

```
0196            memory.signal_strength));
0197         logger(sprintf('%d: Transmitting RTS - %d',ID, MSG_ID))
0198         MSG_ID = MSG_ID + 1;
0199         %after transmitting a RTS
0200         %   node can be either CANDIDATE or in TRANSMIT states
0201         if (memory.state == IDLE)
0202            memory.state = CANDIDATE;
0203         end
0204      else
0205         % wait for current reception to finish.
0206      end
0207   end
0208
0209
0210 case 'NCTS_Timeslot'
0211   PrintMessage('NCTS')
0212   sim_params('set','MAC_PACKET_LENGTH',CTRL_PKT_SIZE);
0213
0214   %Set the current timeslot
0215   memory.currentTS = NCTS_TS;
0216   % Start the Busy Timeslot at the End
0217   Start_DataTimeslot_In(t + CTRL_PKT_SIZE + 2)
0218
0219   %if a node has detected a collision
0220   %   Transmit a NCTS
0221   if (memory.state == VETO)
0222      PrintMessage('S NCTS')
0223      N_CTRL_PKTS_XMIT_RBCAST = N_CTRL_PKTS_XMIT_RBCAST + 1;
0224      Send_Packet(radiostream(struct('ID',ID, 'type',PKT_TYPE_NCTS,...
0225         'MSG_ID',MSG_ID), memory.signal_strength));
0226      logger(sprintf('%d: Transmitting NCTS - %d',ID,MSG_ID))
0227      MSG_ID = MSG_ID + 1;
0228      memory.state = IDLE;
0229   end
0230
0231 case 'Data_Timeslot'
0232   PrintMessage('DATA')
0233   sim_params('set','MAC_PACKET_LENGTH',DATA_PKT_SIZE);
0234
0235   memory.currentTS = DATA_TS;
0236   Start_RTSTimeslot_In(t + DATA_PKT_SIZE + 2)
0237
0238   %if candidate or transmit state
```

```
0239    %   transmit data
0240    if (memory.state == CANDIDATE || memory.state == TRANSMIT)
0241       PrintMessage('S DATA')
0242       memory.state = TRANSMIT;
0243       memory.data_to_transmit = memory.data_to_transmit - 1;
0244       N_DATA_PKTS_XMIT_RBCAST = N_DATA_PKTS_XMIT_RBCAST + 1;
0245       memory.data_xmit_time = t;
0246       MSG_LATENCY = [ MSG_LATENCY ...
0247          (memory.data_xmit_time - memory.rts_xmit_time) ];
0248       memory.rts_xmit_time = -1;
0249       memory.data_xmit_time = -1;
0250       Send_Packet(radiostream(struct('ID',ID, ...
0251          'type',PKT_TYPE_DATA,...
0252          'MSG_ID',MSG_ID,...
0253          'MSG_LENGTH',MESSAGE_LENGTH, ...
0254          'seq_no', (MESSAGE_LENGTH-memory.data_to_transmit), ...
0255          'data_to_transmit',memory.data_to_transmit), ...
0256          memory.signal_strength));
0257       logger(sprintf('%d: Transmitting DATA - %d',ID,MSG_ID))
0258       MSG_ID = MSG_ID + 1;
0259       %All parts transmitted
0260       if (memory.data_to_transmit == 0)
0261          memory.state = IDLE;
0262       end
0263    end
0264 case 'Packet_Sent'
0265    if memory.currentTS == RTS_TS
0266       memory.last_msg_sent = PKT_TYPE_RTS;
0267    elseif memory.currentTS == NCTS_TS
0268       memory.last_msg_sent = PKT_TYPE_NCTS;
0269    elseif memory.currentTS == DATA_TS
0270       memory.last_msg_sent = PKT_TYPE_DATA;
0271    else
0272       error(['Bad current timeslot state'])
0273    end
0274
0275 case 'Packet_Received'
0276    msg=data.data;
0277    switch msg.type
0278       case PKT_TYPE_RTS
0279          if (memory.currentTS == RTS_TS)
0280          % if we are not transmitting in this round
0281          %   we can receive RTS.
```

```
0282          % else
0283          %   radio is in transmit mode
0284          %   ignore packet
0285          if (memory.state == IDLE && ...
0286                (memory.data_to_transmit == 0 || ...
0287                 memory.backoff_rounds > 0))
0288             % check if we have already recvd a RTS this round
0289          if (memory.data_receive_timeout == FALSE)
0290             memory.sender_id = msg.ID;
0291             memory.data_receive_timeout = TRUE;
0292             logger(sprintf('%d: Received RTS - %d',ID,msg.MSG_ID))
0293             CTRL_PKTS_RCVD = [ CTRL_PKTS_RCVD msg.MSG_ID ];
0294             N_CTRL_PKTS_RCVD_RBCAST = N_CTRL_PKTS_RCVD_RBCAST + 1;
0295          else
0296             % Set flag and send out NCTS in the coming NCTS timeslot
0297             logger(sprintf('%d: Collision - Received multiple RTS',ID))
0298             N_CTRL_PKTS_COL_RBCAST = N_CTRL_PKTS_COL_RBCAST + 1;
0299             memory.collision_detected = TRUE;
0300             memory.sender_id = -1;
0301             memory.data_to_receive = 0;
0302             memory.data_receive_timeout = FALSE;
0303             memory.state = VETO;
0304          end
0305          else
0306             %ignore the RTS if we are attempting to transmit
0307             %ERROR_CHECK
0308             if (memory.state ~= IDLE)% && memory.state ~= TRANSMIT)
0309                error(['Bad current state' memory.state ])
0310             end
0311             end
0312          else
0313             error(['Bad current timeslot state - RTS'])
0314          end
0315
0316       case PKT_TYPE_NCTS
0317          if (memory.currentTS == NCTS_TS)
0318          if (memory.state == CANDIDATE)
0319             logger(sprintf('%d: Received NCTS - %d',ID,msg.MSG_ID))
0320             CTRL_PKTS_RCVD = [ CTRL_PKTS_RCVD msg.MSG_ID ];
0321             N_CTRL_PKTS_RCVD_RBCAST = N_CTRL_PKTS_RCVD_RBCAST + 1;
0322             memory.state = IDLE;
0323             memory.backoff_rounds = ceil(rand * MAX_BACKOFF_ROUNDS);
0324          else
```

```
0325            %if we are in TRANSMIT state - ignore NCTS
0326            %neighbors can receive NCTS packet
0327            %But we do not snoop on NCTS.
0328         end
0329         else
0330            error(['Bad current timeslot state - NCTS'])
0331         end
0332
0333      case PKT_TYPE_DATA
0334         if memory.currentTS == DATA_TS
0335         % If we are supposed to receive date OR idle listen for
0336         %       first seq of multi-part DATA
0337         %   receive it
0338         % Else
0339         %
0340         if (memory.state == IDLE)
0341            %in case we didnt receive the RTS
0342            if(memory.sender_id == -1)
0343               %check if this is the first of a multi-part message
0344               if (msg.seq_no == 1)
0345                  memory.sender_id = msg.ID;
0346               else
0347                  %this is the middle of a multi-part message.
0348                  %ignore it
0349               end
0350            end
0351            if (memory.sender_id ~= -1)
0352               logger(sprintf('%d: Received DATA - %d',ID,msg.MSG_ID))
0353               DATA_PKTS_RCVD = [ DATA_PKTS_RCVD msg.MSG_ID ];
0354               N_DATA_PKTS_RCVD_RBCAST = N_DATA_PKTS_RCVD_RBCAST + 1;
0355               memory.data_to_receive = memory.data_to_receive - 1;
0356               memory.data_receive_timeout = FALSE;
0357               %Reset the sender_id when transmission is complete
0358               if memory.data_to_receive == 0
0359                  memory.sender_id = -1;
0360               end
0361            end
0362            elseif (memory.state ~= TRANSMIT)
0363      logger(sprintf('%d: Received unknown Data transmission %d.',...
0364            ID, msg.MSG_ID));
0365            else
0366               % node was in transmit state
0367               % radio in transmit mode.
```

```
0368                % ignore data recvd
0369             end
0370          else
0371             error(['Bad current timeslot state - DATA'])
0372          end
0373
0374       otherwise
0375          error(['Bad pkt type: ' msg.type])
0376    end
0377
0378 case 'Collided_Packet_Received'
0379    %PrintMessage('PC')
0380    msg = data.data;
0381    %disp(sprintf('%d: Received collided packet',ID));
0382    switch memory.currentTS
0383       case RTS_TS
0384       switch memory.state
0385       case CANDIDATE
0386          % its possible to exist - just transited
0387          % radio in transmit mode - so ignore collision
0388       case IDLE
0389          % if we are not in transmit mode
0390          %   => we can detect collisions
0391          if (memory.data_to_transmit == 0 || memory.backoff_rounds > 0)
0392             logger(sprintf('%d: Collision - RTS',ID))
0393             N_CTRL_PKTS_COL_RBCAST = N_CTRL_PKTS_COL_RBCAST + 1;
0394             memory.collision_detected = TRUE;
0395             memory.state = VETO;
0396          else
0397             % in transmit mode
0398          end
0399       case TRANSMIT
0400          % its possible to exist - from previous round
0401          % radio in transmit mode - so ignore collision
0402       otherwise
0403          logger(sprintf('%d: RTS_TS: Bad state - %d',ID, memory.state))
0404          error(['Bad state in RTS_TS, ID: ' ID])
0405       end
0406
0407       case NCTS_TS
0408          switch memory.state
0409             case CANDIDATE
0410                % multiple transmitters detected - back off
```

```
0411                logger(sprintf('%d: Collision - NCTS',ID))
0412                N_CTRL_PKTS_COL_RBCAST = N_CTRL_PKTS_COL_RBCAST + 1;
0413                memory.collision_detected = TRUE;
0414                memory.backoff_rounds = ceil(rand * MAX_BACKOFF_ROUNDS);
0415                memory.state = IDLE;
0416             case IDLE
0417                % node waiting for DATA.
0418                % we do not snoop in NCTS phase
0419             case TRANSMIT
0420                % its possible to exist - from previous round
0421                % node is the leader - hence ignore collisions
0422                logger(sprintf('%d: Ignoring Collision - NCTS',ID))
0423                N_CTRL_PKTS_COL_RBCAST = N_CTRL_PKTS_COL_RBCAST + 1;
0424                memory.collision_detected = TRUE;
0425             case VETO
0426                % radio in transmit mode - so ignore collision
0427          end
0428       case DATA_TS
0429          logger(sprintf('%d: Collision - DATA',ID))
0430          N_DATA_PKTS_COL_RBCAST = N_DATA_PKTS_COL_RBCAST + 1;
0431          memory.collision_detected = TRUE;
0432    end
0433
0434 case 'GuiInfoRequest'
0435    if ~isempty(memory)
0436    %disp(sprintf('Memory Dump of mote ID# %d:\n',ID)); %disp(memory)
0437    else
0438    %disp(sprintf('No memory dump available for node %d.\n',ID));
0439    end
0440
0441 case 'Application_Stopped'
0442 % this event is called when simulation is stopped/suspended
0443 sim_params('set_app','N_CTRL_PKTS_XMIT',N_CTRL_PKTS_XMIT_RBCAST );
0444 sim_params('set_app','N_DATA_PKTS_XMIT',N_DATA_PKTS_XMIT_RBCAST );
0445 sim_params('set_app','N_CTRL_PKT_COL',N_CTRL_PKTS_COL_RBCAST);
0446 sim_params('set_app','N_DATA_PKT_COL',N_DATA_PKTS_COL_RBCAST);
0447 sim_params('set_app','N_CTRL_PKTS_RCVD',N_CTRL_PKTS_RCVD_RBCAST);
0448 sim_params('set_app','N_DATA_PKTS_RCVD',N_DATA_PKTS_RCVD_RBCAST);
0449 sim_params('set_app','N_UNIQ_CTRL_PKTS_RCVD',...
0450    length(unique(CTRL_PKTS_RCVD)));
0451 sim_params('set_app','N_UNIQ_DATA_PKTS_RCVD',...
0452    length(unique(DATA_PKTS_RCVD)));
0453 sim_params('set_app','MSG_LATENCY',mean(MSG_LATENCY));
```

```
0454 fclose(fid);
0455
0456 case 'Application_Finished'
0457 % this event is called when simulation is finished
0458 sim_params('set_app','N_CTRL_PKTS_XMIT',N_CTRL_PKTS_XMIT_RBCAST );
0459 sim_params('set_app','N_DATA_PKTS_XMIT',N_DATA_PKTS_XMIT_RBCAST );
0460 sim_params('set_app','N_CTRL_PKT_COL',N_CTRL_PKTS_COL_RBCAST);
0461 sim_params('set_app','N_DATA_PKT_COL',N_DATA_PKTS_COL_RBCAST);
0462 sim_params('set_app','N_CTRL_PKTS_RCVD',N_CTRL_PKTS_RCVD_RBCAST);
0463 sim_params('set_app','N_DATA_PKTS_RCVD',N_DATA_PKTS_RCVD_RBCAST);
0464 sim_params('set_app','N_UNIQ_CTRL_PKTS_RCVD',...
0465    length(unique(CTRL_PKTS_RCVD)));
0466 sim_params('set_app','N_UNIQ_DATA_PKTS_RCVD',...
0467    length(unique(DATA_PKTS_RCVD)));
0468 sim_params('set_app','MSG_LATENCY',mean(MSG_LATENCY));
0469 fclose(fid);
0470
0471 otherwise
0472    error(['Bad event name for application: ' event])
0473 end
0474
0475 S; %%%%%%%%%%%%%%%%%%%%%%% housekeeping %%%%%%%%%%%%%%%%%%%%%%%%%%%%
0476 S;         app_data{ix}=memory;
0477 S; %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
0478
0479 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
0480 %%%%%%%%%%%%%%%%%%                                  %%%%%%%%%%%%%%%%%%%
0481 %%%%%%%%%%%%%%%%%%        COMMANDS                  %%%%%%%%%%%%%%%%%%%
0482 %%%%%%%%%%%%%%%%%%                                  %%%%%%%%%%%%%%%%%%%
0483 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
0484
0485 function b=Send_Packet(data);
0486 global ID t
0487 radio=prowler('GetRadioName');
0488 b=feval(radio, 'Send_Packet', ID, data, t);
0489
0490 function b=Set_Clock(alarm_time);
0491 global ID
0492 prowler('InsertEvents2Q', make_event(alarm_time, 'Clock_Tick', ID));
0493
0494 function b=Start_RTSTimeslot_In(alarm_time);
0495 global ID
0496 prowler('InsertEvents2Q', make_event(alarm_time, 'RTS_Timeslot', ID));
```

```
0497
0498 function b=Start_NCTSTimeslot_In(alarm_time);
0499 global ID
0500 prowler('InsertEvents2Q', make_event(alarm_time, 'NCTS_Timeslot', ID));
0501
0502 function b=Start_DataTimeslot_In(alarm_time);
0503 global ID
0504 prowler('InsertEvents2Q', make_event(alarm_time, 'Data_Timeslot', ID));
0505
0506 function PrintMessage(msg)
0507 global ID
0508 prowler('TextMessage', ID, msg)
0509
0510 function LED(msg)
0511 global ID
0512 prowler('LED', ID, msg)
0513
0514 function logger(msg)
0515 global LOG_LEVEL LOG_SCREEN LOG_FILE fid
0516 if (bitand(LOG_LEVEL, LOG_SCREEN) == LOG_SCREEN)
0517 %         disp('logging to screen');
0518 disp(msg);
0519 end
0520 if (bitand(LOG_LEVEL, LOG_FILE) == LOG_FILE)
0521 %         disp('logging to file');
0522 fprintf(fid, [ msg '\n']);
0523 end
```