私立東海大學

資訊工程研究所

碩士論文

指導教授：呂芳懌 博士

在固定封包速率下並基於頻寬分配之事件觸發式無線感測器網路多路徑壅塞控制

A Rate-allocation Based Multi-path Congestion Control Scheme for Event-Driven Wireless Sensor Networks on Constant Event Packet Rates

研 究 生：邱炫又

中華民國一〇〇年七月

東海大學碩士學位論文考試審定書

東海大學資訊工程學系 研究所

研究生 邱 炫 又 所提之論文

在固定封包速率下並基於頻寬分配之事件觸發式無線感測器網路多路徑壅塞控制

經本委員會審查，符合碩士學位論文標準。

學位考試委員會

召 集 人 ____________ 簽章

委 員 ____________

____

____

指 導 教 授 ____________ 簽章

中華民國 100 年 7 月 14 日

摘要

在本論文中,我們提出一在事件觸發式無線感測器網路(Wireless Sensor Network, WSN)下，基於速率分配及使用高可用性封包傳遞路徑之多路徑壅塞控制方式(multi-path congestion control approach)。本方式包含兩部分,第一部分為建立一以基地台(sink)為根節點，連接 WSN 中各感測器之生成樹，並以其為初始路由路徑。第二部分為藉由使用多路徑及速率分配，來建立一公平之封包傳遞環境。一般來說，新事件的發生會增加網路中的流量，將會使新建立的路由路徑上之節點產生雍塞，尤其是在基地台附近之節點，因而破壞原先流量之平衡並導致封包之丟棄。事實上，若干事件之訊息無法即時地傳遞到 sink，可能會造成決策者因資訊不足而做出錯誤的決策。而在事件結束時，因停止傳送封包，亦將破壞路由上之中間節點的流量平衡。因此，我們動態地調整頻寬以維持各流量間之公平性和善加利用現有資源。實驗結果顯示本方法可有效的改善無線感測器網路的吞吐量、封包延遲及封包遺失率。

關鍵詞：事件觸發，多路徑壅塞控制，公平傳輸，無線感測器網路，速率控制，頻寬調整

Abstract

In this paper, we proposed a rate-allocation based multi-path congestion control approach, called multi-path-congestion control method (MUCON for short), which enforcing high path availability of packet delivery for event-driven Wireless Sensor Networks (WSNs) consists of two parts. The first is constructing a spanning tree to connect all sensor nodes of a WSN to the sink for initial routing. The second is establishing a fair packet forwarding environment by employing multi-path and rate control to deliver packets through routing paths with a balanced manner. Generally, the occurrence of an event will increase net flow, which may congest the nodes on the newly established routing path if the nodes are currently shared by several routing paths, particularly for those nodes near the sink, consequently ruining originally balanced traffic on paths and forcing some event packets to be dropped. In fact, without the information conveyed on the lost packets, users may make an inaccurate decision and react improperly for events. When an event disappears, the surrounding sensor(s) stops transmitting packets. This will again destroy the fairness and unbalance net flows flowing through the co-node routing paths. Thus, to maintain the fairness and balance net flows for a WSN, a downstream node needs to dynamically adjust bandwidths for all its upstream nodes. Experimental results show that this method can effectively improve a WSN' throughputs, shorter end-to-end delays and reduce packet loss rates.

Key Words: Event-driven systems, Multi-path congestion control, Transmission fairness, Wireless sensor networks, Rate control, Bandwidth adjustment.

致謝

首先感謝指導教授呂芳懌博士，在這段期間內，老師在論文及其他方面的指導，讓我體會到做研究應具備的熱忱、淵博的知識、及嚴謹的態度；很高興能在您的指導下完成此篇論文。感謝東海資工陳隆彬老師、楊朝棟老師與林祝興老師在我大學及碩班時期，於專題上及課業上的教導，使我獲益良多。也感謝各位口試委員的建議及指教，使本論文更臻完善。

另外感謝資料庫實驗室志揚、梗延、宏瑋及永倫學長們，在研究方面的指點，使我可以快速融入實驗室的環境；國基和佩勳同學的陪伴與課業上的協助，使我的研究生活更多采多姿；還有逸夫、勝捷、信良、渝新、妙亨學弟妹們的加入，令實驗室多了許多歡笑，在此謝謝你們。

最後感謝我的家人，有你們無怨無悔的付出與支持，讓我可以無後顧之憂的專注在學業上，僅將此文獻給你們。

千言萬語，說不盡的感謝，在這段日子裡，有你/妳們的陪伴，是我最大的榮幸，謝謝你們。

List of Contents

List of Figures

List of Tables

I. Introduction

In recent years, congestion control is one of the most important issues in wireless sensor network (WSN) research [1]. When congestion occurs at a sensor node (a node for short), newly arriving packets will be dropped or enqueued in the node's buffer, causing the facts that upstream nodes have to consume extra energy to retransmit the dropped packets, or packet arrival does not follow the departure sequence. These will not only waste device resources and decrease their lifetime when nodes are powered by batteries [2], but also make the sink to receive discrete and incomplete information, with which users may hard to realize the details of the event and then make an accurate decision to respond to the event in a real time manner. Generally, congestions will degrade network throughputs [1], prolong packet delivery delays [3], and increase packet loss rates [1][3][4], which together are WSN's challenges, particularly for a real-time sensing/monitoring environment.

In an event-driven WSN, sensor nodes detect environmental changes as the occurrence of an event. When there is no events, the load is light. But when events, like fire blaze, earthquake, landslide, or mudflows, occur, a huge number of packets will be suddenly generated. The network traffic from the event points to the sink will be heavy [5]. In a large-scale WSN, hundreds to thousands of sensor nodes are employed to sense events and relay packets [6]. The probability that a node is congested is high, especially for those near the sink [1]. However, if network traffic can be regulated, congestion can be mitigated and packet drop rates will be lowered [1][3][4].

A method to avoid network congestion is load-balanced routing which implies establishing multiple paths [7] between a source node and the sink, and transfers data via the paths. A multi-path routing not only addresses load balancing, and route failure and recovery [8], but also distributes energy consumption and improve packet delivery quality, reliability and throughputs [8][9][10]. Constructing a spanning tree to route packets has been used by [10][11]. In fact, if we can construct such a tree as the initial routing tree for a WSN during its system startup, then when events occur, packets can be delivered immediately without sending a route request packet to establish a path from the source to the sink before packets can be sent, consequently shortening packet delivery delays. Furthermore, when the paths are congested, if we can establish alternate paths, i.e., a multi-path routing, then the packet delivery performance can be further improved [11].

Therefore, in this paper, we propose a rate-allocation based congestion control approach,

called multi-path-congestion control method (MUCOM for short), which can be deployed by an event-driven WSN to construct multiple paths for a congested node and regulate packet flows for the paths so as to improve performance of packet delivery. Experimental results show that the MUCOM not only reduces packet drop rates and congestion probability, but also dramatically mitigates the waste of network bandwidth and improves performance of packet transmission.

The key contributions of this study are as follows.

1) In the MUCOM, when a node *N* is congested, we do not decrease the data rates of all source nodes of which the routing paths go through *N*. Instead, we establish alternate links for *N* to acquire additional bandwidth to transfer congested data packets.
2) We use rate-based bandwidth control to adjust the bandwidth for each link when an event occurs or disappears so as to effectively improve the total utilization rate for a WSN.
3) We construct a spanning tree as the initial routing paths for source nodes so that when an event occurs, source node can delivers data packets immediately, i.e., omitting the delay due to route discovery.

The rest of this paper is organized as follows. Section II describes the related work of this study. Section III introduces the algorithms and processes of the MUCOM. Its simulation results are presented and discussed in Section IV. Section V concludes this paper and outlines our future research.

II. Related Work

So far, several congestion control approaches [1][3][5][6][12][13] and routing protocols [10][11] have been proposed. Hybrid Congestion Control Protocol (HCCP) [1] integrated a buffer-based congestion control method and a rate-based congestion control method to control congestions. When the net flow flowing through a node *N* exceeds *N*'s buffer size, the HCCP regulates *N*'s data rate to mitigate packet drop rates. However, the authors did not describe how to adjust the system-wide rates, but pointing out that it is insufficient for congestion control if only buffer and packet delivery rates are considered. In the MUCOM, the bandwidth required by the source nodes along a path is always equal to or less than the available bandwidth of the path so that the influence of buffer size is small and can be even ignored.

Priority-based Congestion Control Protocol [3] predicted the probable congestion by collecting packet service time and packet inter-arrival time, with which a scheduler was developed to control network bandwidth. Fairness-Aware Congestion Control (FACC) [5] categorized intermediate relay nodes into near-source nodes and near-sink nodes, and used different strategies to assign appropriate fair rates to them to avoid congestion and save energy. Near-source nodes maintain a per-flow state by monitoring channel business and allocate an approximately fair rate to each passing flow. Near-sink nodes are installed a lightweight probabilistic dropping algorithm based on queue occupancy and hit frequency. If the queue occupancy is higher than a predefined upper bound, the arrival packet will be dropped and the rate of all passing flows will be reduced. If the queue occupancy is now between the upper and lower bounds, the data rate of the corresponding source node will be adjusted. However, when the scale of a WSN is small, it is unclear that a node is a near-source or a near-sink node.

Monowar et al. [6] employed a queuing model composed of many queues to handle different types of data packets and a classifier provisioned in network layer to classify heterogeneous traffic. It used packet service ratio denoted on average packet service rate and packet scheduling rate to detect congestions. Sankarasubramaniam et al. [12] proposed an Event-to-sink reliable transport (ESRT) which defines network states, the corresponding operations of a state, current state, and a reliability indicator denoted by the realistic number of received packets and the desired number of received packets. By frequently updating current state and reliability indicator at the sink, the ESRT could accurately identify current

network state, and then executes the corresponding operations, like adjusting source nodes' reporting frequencies. However, all the computations are performed at the sink, of which the load is high.

Congestion detection and avoidance (CODA) [13] detected buffer utilization of nodes, and current and previous channel loading rates to predict degree of congestion for receiver nodes. The CODA also employed sampling monitors in an appropriate time to reduce energy consumption. When the node is congested, it backpressures all upstream nodes to adjust data transmission rates or drop packets. As with the CODA, the MUCOM adjusts bandwidth for each node along a routing path when necessary. The adjustment activates are propagated from the sink toward upstream nodes.

Lou [10] developed a distributed 'N-to-1' multi-path finding protocol that used flooding approach to generate a typical spanning tree from the sink, and find multiple node-disjoint paths from sensor nodes simultaneously, in order to reduce the latency of path establishment. It also offered a packet salvaging strategy to improve the reliability of packet delivery. But this approach needs path information from source to the sink so that the loading of nodes is heavy. EAMTR [11] is a light weighting routing protocol which generates multiple trees for a source node, and each node selects the least congested routes based on Link Quality Indication, the index defined by 802.15.4 standard, to send packets to the sink. It improves the reliability of links by providing redundant paths. But the generation of multiple trees is a heavy burden for those highly-occurred event areas.

The abovementioned methods are proposed under their specific environments. Each has its own advantages and disadvantages. However, the congestion control methods except the FACC did not address fairness of packet transmission among routing paths. Generally, these methods start when congestion occurs. In the MUCOM, the flow control between a node and its upstream nodes begins only when events occur and terminate. When congestion occurs on a node N, N starts finding multiple paths to relieve congestion. The relief is performed by all sensor nodes in a distributive manner, instead of by the base station. This can reduce the computation burden of the base station and network load of its surrounding nodes.

III. The Proposed Scheme

In the study, we assume that: 1) A node is linked with at least one neighbor node, i.e., each node is reachable from the sink(s) so that packets can be successfully delivered to the sink(s); 2) A node knows where the sink is and the relationships between itself and all its upstream nodes and downstream nodes; 3) In a WSN, a link's initial bandwidth B, also called default bandwidth, is the maximum bandwidth of the link and known beforehand; 4) The data rates Rs generated by different source nodes are the same, i.e., R is a constant which is less than or equal to the default bandwidth of a link; 5) The sizes of data packets are the same.

Here, a *link* is defined as the direct connection between two nodes, e.g., nodes a and b, without any immediate node located between them. A *path* consists of many links, i.e., at least k nodes and $k+1$ links located between a and b, $1 \leq k$.

3.1. System phases

In this study, we define three operational phases for data delivery between a source node and the sink, including link establishment, rate allocation and adjustment, and the multipath congestion alleviation. The link establishment phase is to establish a spanning tree from root node (the sink) to all sensor nodes for a WSN so that each source node of the WSN has its own initial routing path to deliver packets.

The sink on receiving the first packet of an event, which is a piggybacked data packet [14] called start piggybacked packet (SP_packet for short) issued by a source node, or the last packet of an event, also a piggybacked packet called end piggybacked packet (EP_packet for short), starts adjusting the bandwidths for all its upstream nodes.

Once a routing path, e.g., P, is congested at a node N, i.e., the bandwidth allocated to N is lower than N's current data rate, which occurs when a new event is sensed by N or extra bandwidth is required by N's upstream nodes (by issuing an SP_packet to N), the multipath congestion alleviation phase is then triggered to establish an additional downlink for N.

3.2. The link establishment phase and configuration table

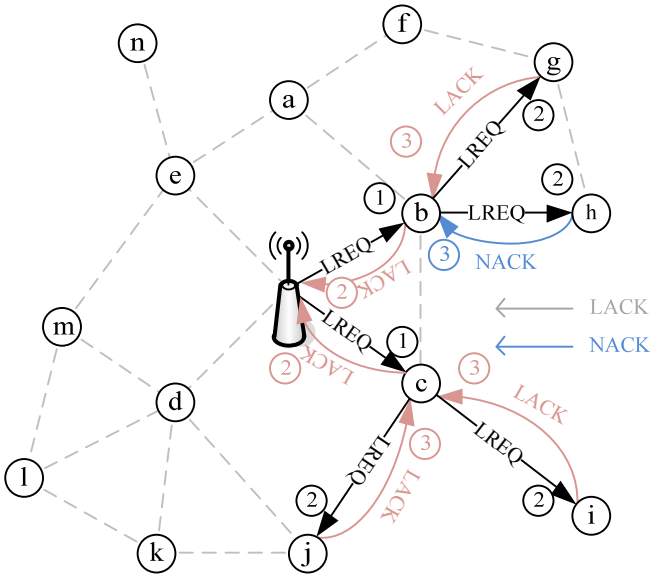
In this study, a spanning tree is established by using a flooding-based approach from downstream nodes toward upstream nodes when the WSN being considered starts up. The sink first broadcasts an LREQ packet which as shown in Figure 1 contains the message type, i.e., LREQ, and the SenderID which is the ID of the sink. Initially, all nodes of the WSN are full of energy.

LREQ	SenderID

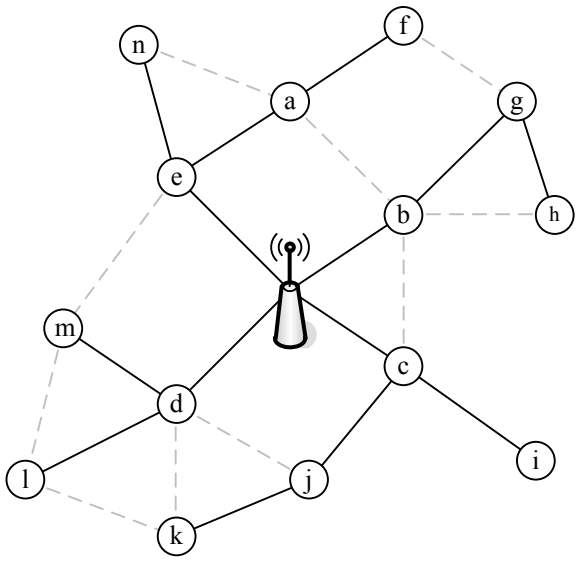
Figure 1. Fields of an LREQ packet

If any neighbor node *Q* which currently has already established a link, i.e., an downlink, with other node, replies a negative response NACK and discards the LREQ since in this phase one node can establish only one downlink. Otherwise, it replies an LACK (the abbreviation of Link ACK), broadcasts the LREQ to all its neighbors to continue constructing links for the tree, and discards all later receiving LREQ packets. The process repeats until no more LREQ is sent, i.e., all active nodes are linked together as a spanning tree. Note that in this phase only the node *N* that has established a downlink and has no upstream nodes can issue an LREQ packet, and *N* may receive LACKs from several *Q*'s. Hence, *N* may have several uplinks.

We assume that the neighbor nodes of the sink are all connected to the sink after the link establishment phase because each of them receives the LREQ issued by the sink and replies an LACK. Figures 2a and 2b respectively show a partial process of the link establishment phase and the established spanning tree after this phase.



(a) A partial process of the link establishment phase



(b) The established spanning tree after the phase

Figure 2. The link establishment phase

With the MUCOM, a node *N* records the information of the concerned nodes and links in a table, called configuration table, which as shown in Table 1 includes *NodeID*, *LinkType*, *LinkedNodeID*, *LinkNum* and *Bandwth*, in which *NodeID* records the node's unique ID, i.e., *N*. *LinkType* representing the type of a link connecting one of *N*'s neighbor node, e.g., *S* and *N* can be one of the three values, *GN*, *UP* and *DW*, which respectively indicate that *N* is a source node, *S* is an upstream node of *N* and a downstream node of *N*. If *N* has more than one connection, i.e., having many upstream nodes and downstream nodes, we create multiple tuples to record each of them. *LinkNodeID* lists the ID of the concerned node, i.e., *S*. *LinkNum* field records the order an *N*'s downlink is established. Note that only *N*'s downlinks are considered. Hence, it is empty if the corresponding node is an upstream node or a source node, i.e., *LinkType* = *UP* or *GN*. Its value will be described later. *Bandwth* field records the bandwidth that a downstream node D_i allocates to *N* if *LinkType* = *DW* or *N* allocates to the corresponding upstream node U_j if *LinkType* = *UP*, and the default data rate if *LinkType* = *GN*. In this phase, the *bandwth* field values of all established tuples are set to default bandwidth.

Table 1. Node *a*'s configuration table after the link establishment phase (see Figure 2b)

NodeID	**LinkType**	**LinkNodeID**	**LinkNum**	**Bandwth**
a	UP	f	--	default
a	DW	e	1	default

Table 2. Node *e*'s configuration table after the link establishment phase (see Figure 2b)

nodeID	**LinkType**	**LinkNodeID**	**LinkNum**	**Bandwth**
e	UP	a	--	default
e	UP	n	--	default
e	DW	Sink	1	default

Tables 1 and 2 respectively list the configuration tables for nodes *a* and *e* in the topology illustrated in Figure 2b. Node *a* is one of node *e*'s upstream nodes. The other is node *n*. The field *Bandwth* in each tuple will be updated during the rate allocation and adjustment phase.

3.3. The rate allocation and adjustment phase

A data packet consists of two parts, the sensed data and metadata. The sensed data may be degree of brightness, temperature, humility, or the shake of a detected object depending on what type of sensor the node is equipped. The metadata contains the common information about the data packet *p,* including the coordinates of the node generating *p*, e.g., node *N,* and *N*'s neighbor nodes, the time *p* is sensed, packet sequence, *N*'s residual energy and other information used to recognize the packet. But data packets are not the focus of this paper.

In the rate allocation and adjustment phase, the two piggybacked packets, i.e., SP_packet and EP_packet as stated above, are created for each event to respectively represent the beginning and the end of an event. Figure 3 shows their format including *EventType*, *SrcID*, *SenderID*, *RecverID* and *Bandwth* fields. Here, EventType shows that the packet *p* is an SP_packet or EP_packet. SrcID keeps the source node ID. SenderID and RecverID records the ID of the node that sends/receives *p*. For example, if an SP_packet generated by a source node *q* (i.e., *SrcID* = *q*) is sent by an intermediate node *s* to node *t*, the value of SenderID is *s* instead of *q* and the value of RecverID is *t*. The *Bandwth* field carries the default data rate of *q* if *p* is an SP_packet or the allocated bandwidth if *p* is an EP_packet so that a receiving node can accordingly calculate the total data rate for all its upstream nodes for the next time slot.

Figure 3. Extra fields of a piggybacked packet as the first or the last packet of an event

The rate allocation and adjustment phase begins when the first event of the concerned WSN occurs. Each time, nodes discover that there are events, they send SP_packets to tell nodes along the routing paths to increase their upstream-node data rates, and indicate that regular data packets of the new events will soon arrive. When a node *N* on the path overflows,

it sends an MREQ packet to request establishing an alternate path so that packets can safely arrive at the sink. Once one of node *N*'s upstream events disappears, due to previous rate adjustment, *k* links of *N* may be no longer required where *k* is the smallest integer, $1 \le k \le n$, satisfying the condition that

$$\sum_{i=1}^{n} B_{Di}^{t} - \sum_{i=k}^{n} B_{Di}^{t} \ge (B_{N}^{t+1} + \sum_{j=1}^{m} B_{Uj}^{t+1}) \qquad (1)$$

in which *n* and *m* are respectively the numbers of *N*'s immediate downstream and upstream nodes, B_{Uj}^{t} is the bandwidth allocated to the link between upstream nodes U_j and *N* in timeslot *t*, B_{Di}^{t} is the allocated bandwidth of the link D_i between *N* and its downstream node D_i, and B_N is the default data rate of *N* if *N* is now an active source node. $B_N = 0$ if *N* is a non-source node or currently an inactive source node that only relays packets for its upstream nodes. Generally, $\sum_{i=1}^{n} B_{Di}^{t}$ is *N*'s output bandwidth and $B_{N}^{t+1} + \sum_{j=1}^{m} B_{Uj}^{t+1}$ is the input data rate of *N*. *Eq. (1)* implies $l_k, l_{k+1}, ..., l_{n-1}, l_n$ can be released. We follow the sequence: $l_n, l_{n-1}, ..., l_{k+1}, l_k$ to release the links. *N* delivers *n-k+1* EP_packets though the *n-k+1* downlinks $l_k, l_{k+1}, ..., l_{n-1}, l_n$ that are links between *N* and *n-k+1* downstream nodes $D_k, D_{k+1}, ..., D_n$. D_i on receiving the packet removes the corresponding uplink by deleting *N*'s corresponding tuple from D_i's configuration table, $i = k, k+1, ..., n-1, n$. *N* also deletes the corresponding *n-k+1* tuple from its configuration table. If *Eq. (1)* is ">" instead of "≥", implying the bandwidth of l_{k-1} needs to be reduced, an additional EP-packet with *Bandwth* = $B_{Dk-1}^{t} - [\sum_{i=1}^{n} B_{Di}^{t} - \sum_{i=k}^{n} B_{Di}^{t} - (B_{N}^{t+1} + \sum_{j=1}^{m} B_{Uj}^{t+1})]$ will be sent through l_{k-1}.

We set a timer at the sink to count the length of a time slot. Basically, the sink receives packets including data and piggybacked packets from its upstream nodes. When the underlying time slot *t* expires, if sink has received at least one piggybacked packet in current timeslot, it sends an RACK packet which as shown in Figure 4 contains the *SenderID* = the sink, *RecverID* = *U,* and *Bandwth* = the default bandwidth as the response to all piggybacked packets, SP_packets and/or EP_packets, received in *t.* Otherwise, it does not send any RACK to its upstream nodes to reduce control traffic.

RACK	SenderID	RecverID	Bandwth

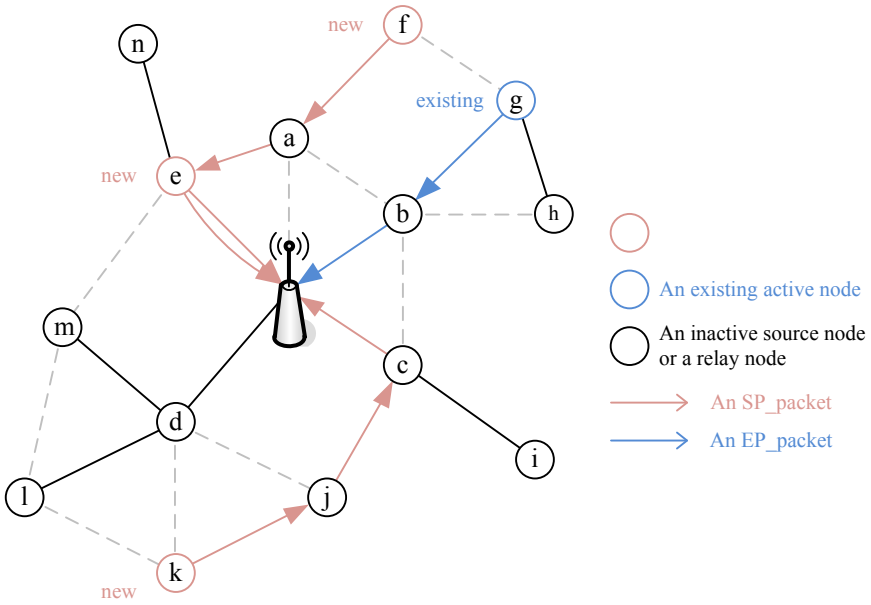
Figure 4. Fields of an RACK packet

A node, e.g., N, on receiving an RACK packet at timeslot t allocates the bandwidth B_{Ui}^{t+1} to its upstream node U_i for time slot $t+1$ where

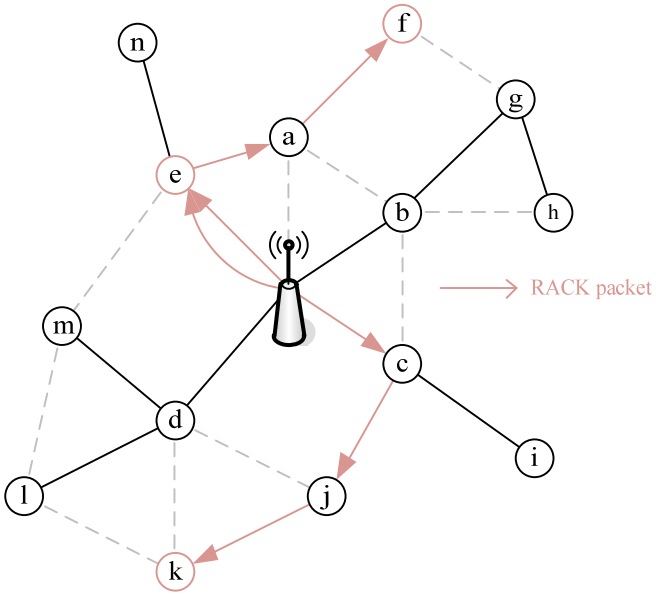
$$B_{Ui}^{t+1} = \min(B_{Ui,default}, \frac{B_{Ui}^{t}}{B_N^t + \sum_{j=1}^{m} B_{Uj}^t} \times \sum_{i=1}^{n} B_{Di}^{t+1}) \qquad (2)$$

implying this is a frequency division scheme [15] where $B_{Ui,default}$ is the default bandwidth of the link between U_i and N, m is the number of N's active upstream nodes, n is the number of N's downstream nodes allocating bandwidths to N, B_{Di}^{t+1}, the bandwidth conveyed on the received RACK, is the bandwidth that N's downstream node D_i allocates to N for $t+1$, B_{Ui}^t is the bandwidth allocated to the link between N's upstream node U_i and N at t (if U_i is a newly established link, B_{Ui}^t is the default bandwidth) and B_N^t is the default data rate if N is now an active source node. Otherwise, $B_N^t = 0$. An active source node is a node discovering an event. It remains active until the termination of the event. During the period of time, N send data packets to the sink periodically. N also relays packets for its upstream nodes, no matter it is an active source node or not. The data rate generated by an active source node is a constant.

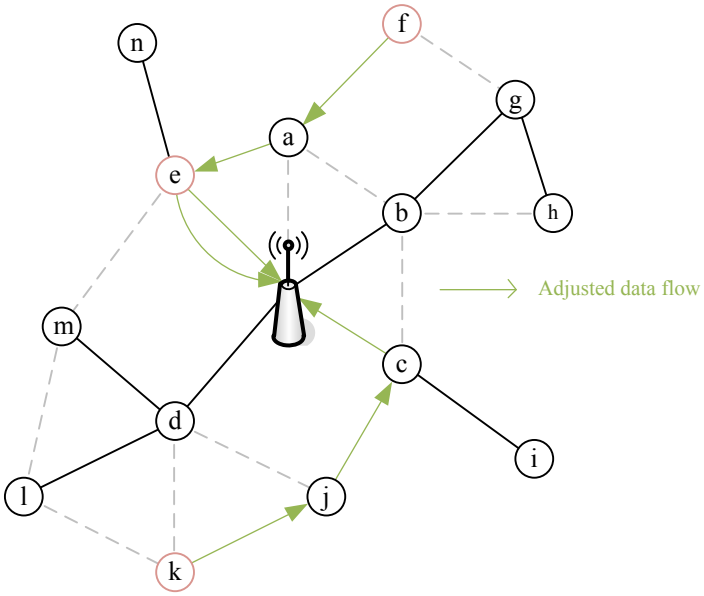
Figure 5 illustrates an example of the network derived from Figure 2b in which nodes e, f, and k are new active source nodes, and node g is an existing active source node. Figures 6a and 6b illustrate the sequence of activities of the rate allocation and adjustment phase for nodes e, f and g.


new
f
existing
g
n
a
new
e
b
h
m
c
d
i
l
j
k
new
An existing active node
An inactive source node
or a relay node
An SP_packet
An EP_packet

(a)

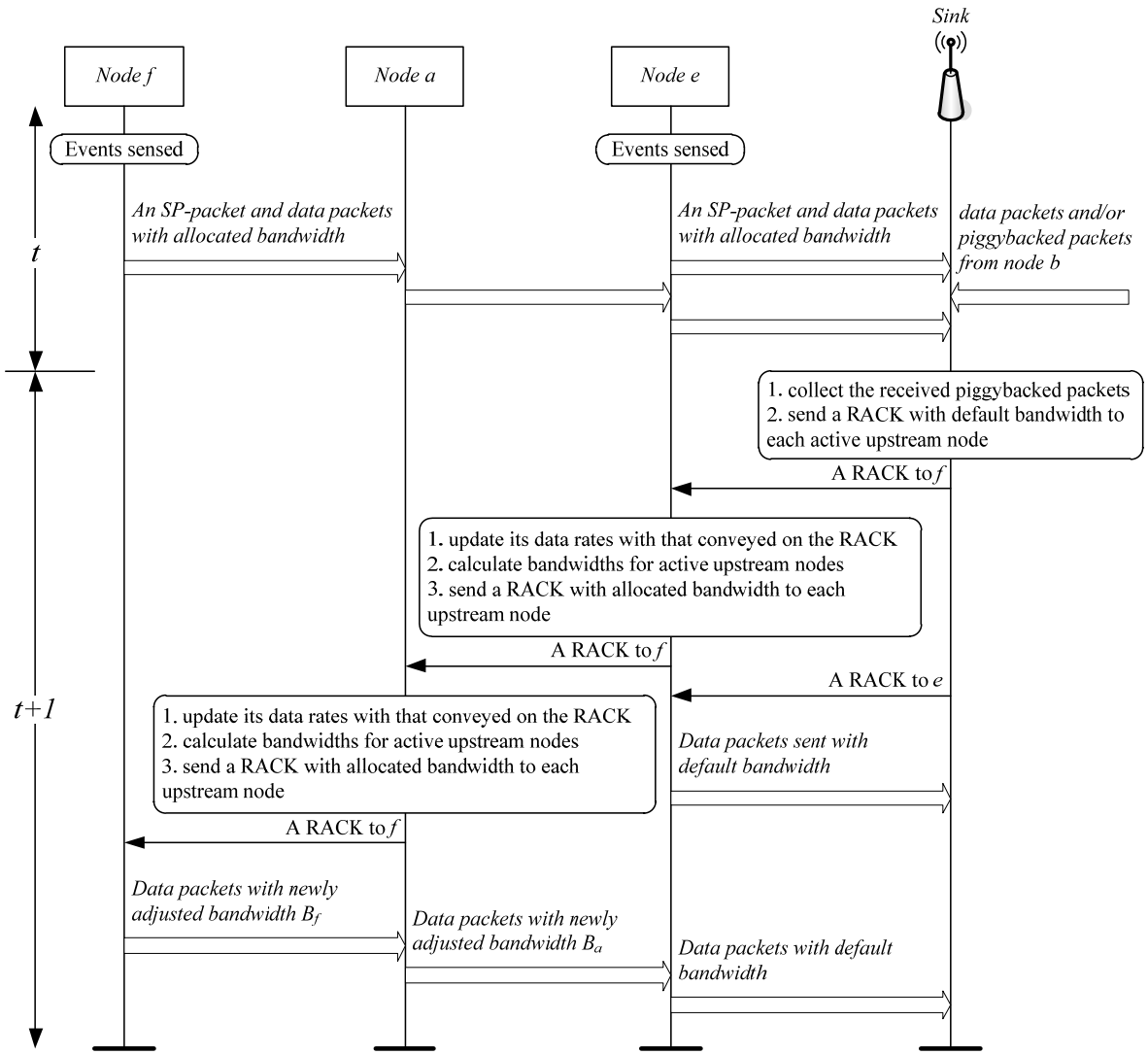

f
n
g
a
e
b
h
RACK packet
m
c
d
i
l
j
k

(b)

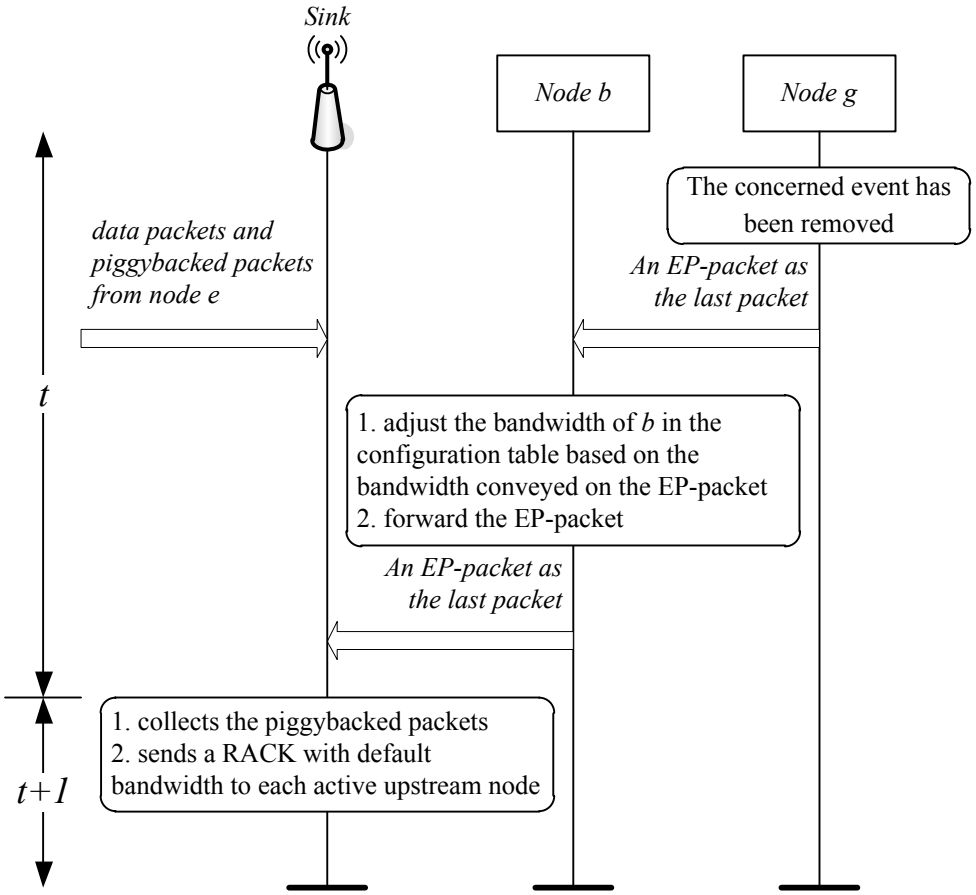


(c)

Figure 5. An example network for illustrating the process of the rate allocation and adjustment phase. (a) a network topology derived from Figure 2b, in which node *g* is an existing active source node, but its event has just disappeared. Nodes *e*, *f*, and *k* discover new events, and others are inactive nodes or non-source nodes; (b) the sink and intermediate nodes deliver RACKs to adjust bandwidth for each node; (c) nodes *e*, *f*, and *k* follow their allocated bandwidths to deliver data packets



(a) The procedure for newly discovered events by nodes *f* and *e*



(b) The procedure for a disappearing event

Figure 6. The sequence diagrams of rate allocation and adjustment phase

Table 3. Node *e*'s configuration table after the rate allocation and adjustment phase (see Figure 2b)

NodeID	**LinkType**	**LinkNodeID**	**LinkNum**	**Bandwth**
e	UP	a	--	default bandwidth
e	UP	n	--	default bandwidth
e	DW	Sink	1	default bandwidth
e	GN	e	--	default data rate

Table 3 lists the configuration table for node *e* in the topology illustrated in Figure 2b after the rate allocation and adjustment phase. The last tuple is added since *e* is now an active source node. Node *a*'s configuration table after this phase is still the one shown in Table 1.

3.4. Multipath congestion alleviation phase

The multipath congestion alleviation phase begins when a node, e.g., node *N*'s, allocated

or newly adjusted bandwidth $\sum_{i=1}^{n} B_{Di}^{t+1}$ for timeslot *t+1* is lower than its total data rate $B_N^t + \sum_{j=1}^{m} B_{Uj}^t$ in *t+1*. Then, *N* broadcasts a MREQ packet which as shown in Figure 7 includes SenderID, i.e., *N*, and

$$\text{BandReq} = (B_N^t + \sum_{j=1}^{m} B_{Uj}^t) - \sum_{i=1}^{n} B_{Di}^{t+1} > 0 \qquad (3)$$

MREQ	SenderID	BandReq

Figure 7. Fields of a MREQ packet

Any neighbor node, e.g., node *Q,* on receiving the MREQ packet looks up *Q*'s configuration table. If *N* is in the *LinkNodeID* field of a tuple, implying that the link already exists, *Q* drops the packet. Otherwise, *Q* replies an MACK packet, which as shown in Figure 8 includes a sender ID, i.e., *Q*, a receiver ID, i.e., *N*, indicating the reply of the MREQ packet, i.e., the MACK, is sent to *N*, and BandAvail = default bandwidth since current no traffic flows through the link that will be established link between *Q* and *N*. *N* on receiving the first MACK packet, e.g., sent by *Q*, establishes a link l_{NQ} between itself and *Q*, assigns B_{NQ} = min(*BandReq, BandAvail*) to l_{NQ}, and replies an MEST, of which the format is shown in Figure 9, with *Bandwth* field = B_{NQ} to tell *Q* the bandwidth of the newly established link l_{NQ}, denoted by B_{NQ}. Like that in the link establishment phase, *N* discards the later receiving MACK packets. However, if *BandReq* > *BandAvail* implying *N* is still congested. *N* issues another MREQ with *BandReq* = *BandReq* – *BandAvail* to request another downlink. The process repeats until *BandReq* = min(*BandReq, BandAvail*) which means *BandAvail* ≥ *BandReq*, i.e., *N* is no longer congested. In Figure 9, *LinkNum* is the sequence the downlink is established.

MACK	SenderID	RecverID	BandAvail

Figure 8. Fields of a MACK packet

MEST	SenderID	RecverID	LinkNum	Bandwth

Figure 9. Fields of a MEST packet

Each time when a data packet or a piggybacked packet should be delivered by *N,* and *N* has more than one downlink, *N* sends the packet to its downstream node D_i with the

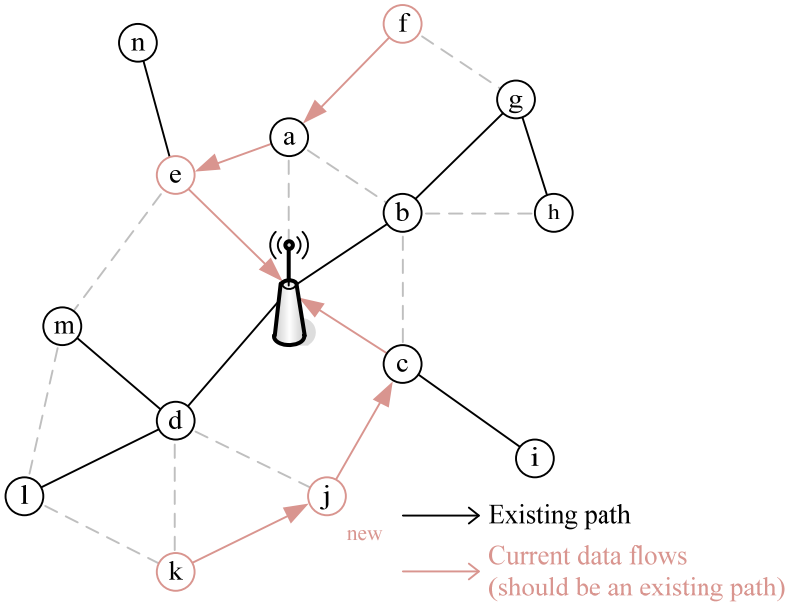
probability P_{Di}^{t}

$$P_{Di}^{t} = \frac{B_{Di}^{t}}{\sum_{j=1}^{n} B_{Dj}^{t}} \qquad (4)$$

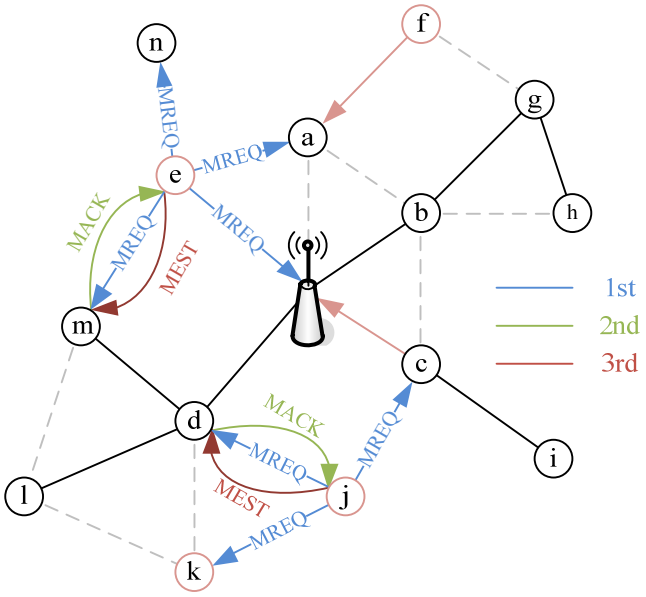
where n is the number of downstream nodes of N, and B_{Di}^{t} is the bandwidth allocated to the link between N and D_i by D_i for time slot t. Note that the spanning tree path and the paths established beforehand are still in use.

The creation and deletion of multiple links is based on the field value of *LinkNum*, which is generated in the link establishment phase and multipath congestion alleviation phase. In the link establishment phase, when a downlink of N is established, *LinkNum* of the corresponding tuple as states above is set to *1*, representing it is a link of the spanning tree, called N's primary link, which cannot be deleted, even though its traffic is zero. In the multipath congestion alleviation phase, *LinkNum* of which the value ranged from *2* to n is used to record the order that a new link is established. When some links due to zero traffic are no longer required by N, we delete the links following the reverse order, from n to *2*, of *LinkNum.*

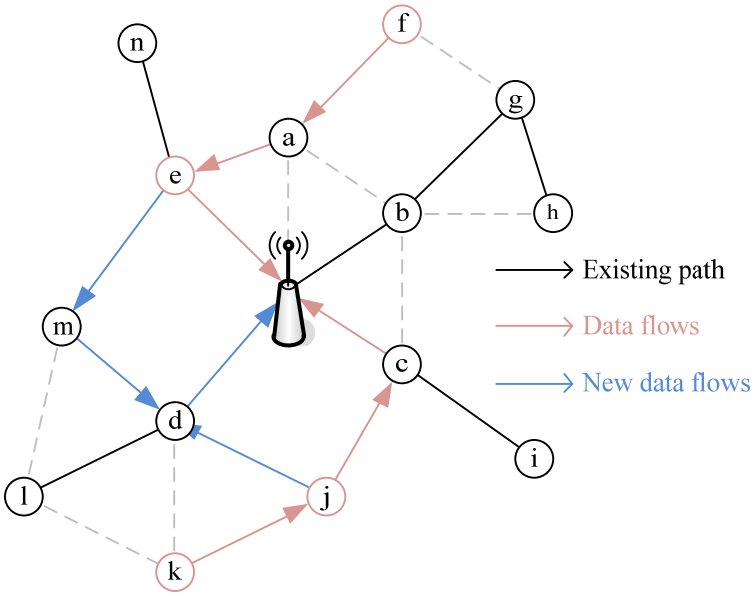
Since we assume that the data rate of an event sensed by node N cannot exceed the default bandwidth of the link between N and one of its downstream nodes so that we only need to create an additional downstream link for N once an event is discovered by N. However, it is possible that N's h upstream nodes also discover events, $h \geq 1$. Then, *BandReq* conveyed on an MREQ is higher than the default bandwidth of a link. That is why N may continuously create k downlinks, $k \geq 1$. Figure 10 illustrates an example network derived from Figure 5. In this example network, nodes e and j individually broadcast an MREQ packet and receive MACK packets. Only m replies an MACK to e, and d replies MACK to j, e and j consequentially establishing the links, $e \rightarrow m$ and $j \rightarrow d$. Now node e (node j) can also deliver packets through the alternate paths $e \rightarrow m \rightarrow d \rightarrow Sink$ ($j \rightarrow d \rightarrow Sink$). Figure 11 shows the sequence diagram of the multipath congestion alleviation phase.


f
n
g
a
e
b
h
m
c
d
i
l
j
new
k
Existing path
Current data flows
(should be an existing path)

(a)


f
n
g
MREQ
a
MREQ
e
b
h
MACK
MREQ
MREQ
MEST
1st
2nd
3rd
m
c
d
MACK
MREQ
MREQ
i
MEST
l
j
MREQ
k

(b)



(c)

Figure 10. Network activities on the example topology shown in Figure 5 in its multipath congestion alleviation phase (a) a derived network topology from Figure 5c in which *e*, *f*, *j* and *k* transfer data packets to their downstream nodes, and *e* and *j* are congested; (b) *e* and *j* broadcast MREQs, *e* receives an MACK from *m* and *j* receives an MACK from *d*, and *e* (*j*) chooses *m* (*d*) as an alternate link; (c) *e* uses path $e \rightarrow Sink$ and $e \rightarrow m \rightarrow d \rightarrow Sink$, and *j* uses path $j \rightarrow c \rightarrow Sink$ and $j \rightarrow d \rightarrow Sink$ to transfer data packets.

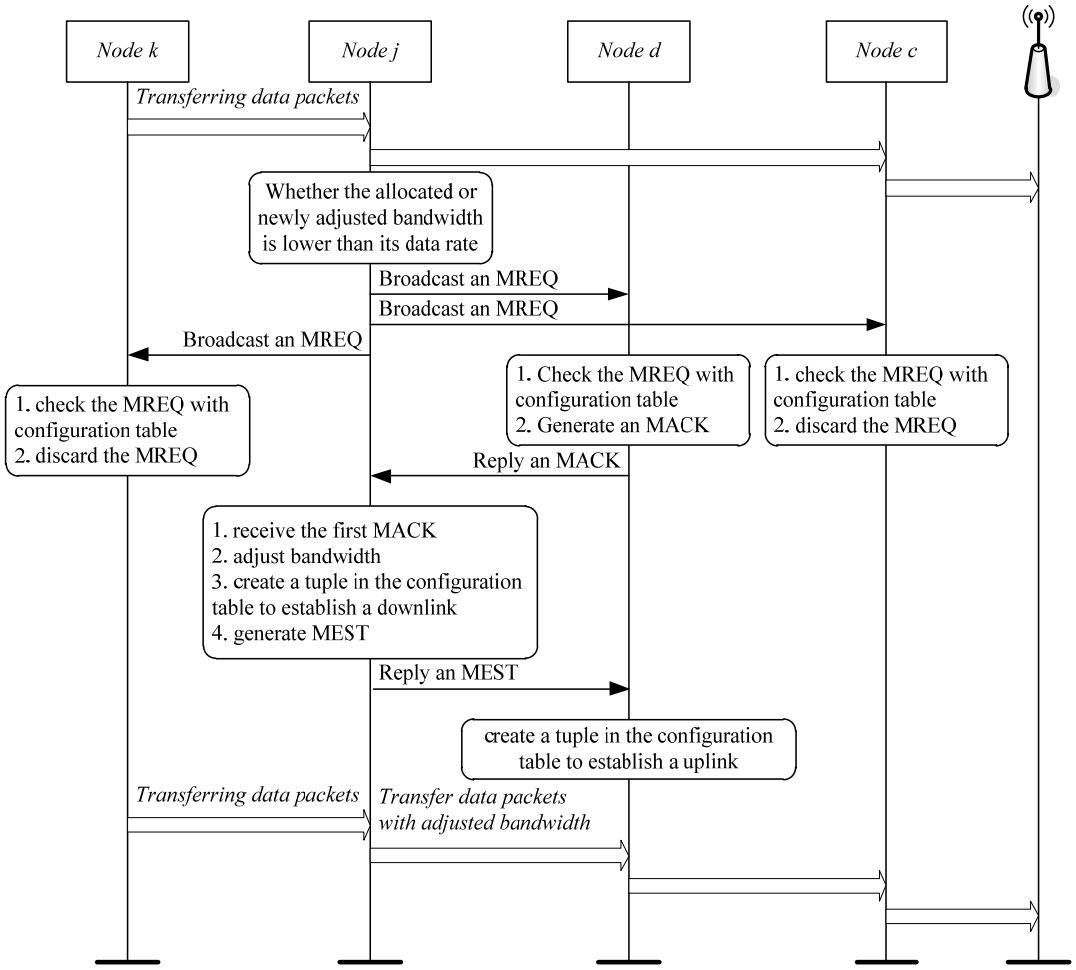


Figure 11. The sequence diagram of multipath congestion alleviation stage

Table 4. Node *e*'s configuration table after the multipath congestion alleviation phase (see Figure 2b)

NodeID	**LinkType**	**LinkNodeID**	**LinkNum**	**Bandwth**
e	UP	a	--	default bandwidth
e	UP	n	--	default bandwidth
e	DW	Sink	1	default bandwidth
e	DW	m	2	$B_{\overline{em}}$
e	GN	e	--	default data rate

Table 4 lists the configuration table for node *e* in the topology illustrated in Figure 2b after the multipath congestion alleviation phase. The fourth tuple is added since *m* is now a new downlink of *e*, and with allocated bandwidth $B_{\overline{em}} = \min(BandReq, BandAvail)$ where

BandReq and *BandAvail* are respectively calculated by node *e* and *m*. Node *a*'s configuration table after this phase is still the one shown in Table 1.

3.5. Algorithms

The algorithms proposed in this scheme can be classified into function of the sink *S* and those of a node *N* other than the sink. The former only includes *Sink()*, whereas the latter contains 13 functions, including *Node()*, *eventStart()*, *eventStop()*, *rcvPacket()*, *rcvDataAtNode()*, *rcvCtrlAtNode(), linkCreation()*, *linkDeletion()*, *transferData(), rcvLREQ(), rcvRACK(), rcvMREQ(),* and *rcvMACK()*. The two parts constitute operations of a concerned WSN. *Sink()* in *S* and *Node()* in *N* are the main programs of *S* and *N*, respectively. Figure 11 illustrates the program structure of *N*. *eventStart()* (*eventStop()*), involved when *N* starts (stops) sensing an event *E*, further calls *linkCreation()* (*linkDeletion()*) to create an additional link (delete multiple links) when *N* overflows (some downlinks of *N* are no more required). *rcvPacket()*, involved when *N* receives a packet *p*, invokes *rcvDataAtNode()* (*rcvCtrlAtNode()*) if *p* is a data packet (control packet). *transferData()* is called by *Node()* and *rcvDataAtNode()* to transfer data packets though a path or paths. When *N* receives an LREQ, a RACK, an MREQ and an MACK packets, *rcvCtrlAtNode()* will call *rcvLREQ(), rcvRACK(), rcvMREQ()* and *rcvMACK()*, respectively, to perform the corresponding tasks.

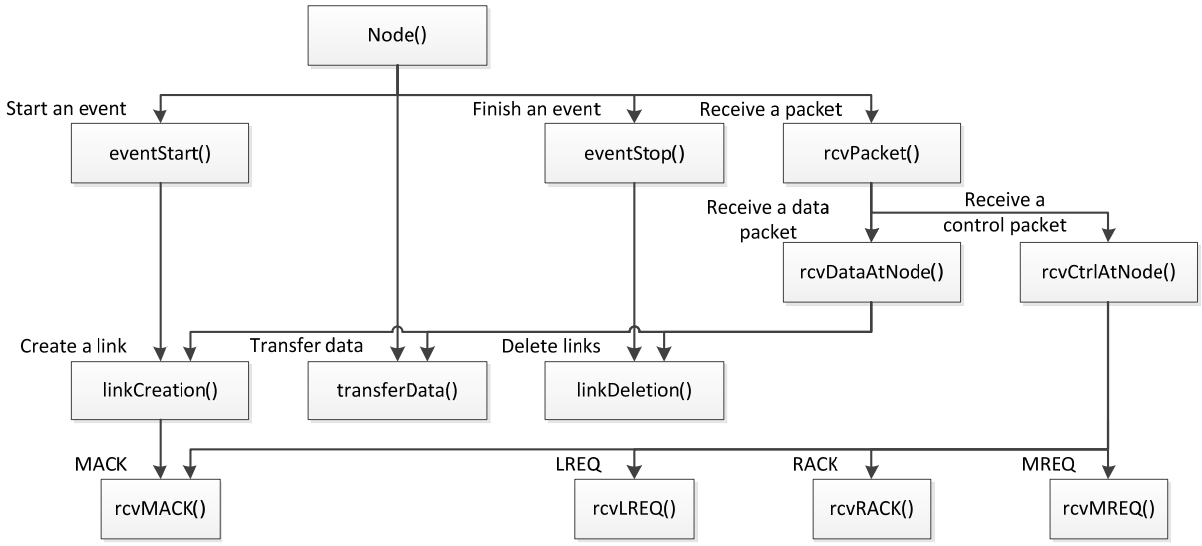


Figure 12. The program structure of a node other than the sink

At the beginning of the system, the sink *S* activates *Sink()* which broadcasts an LREQ packet and sets a timer to the time period T_P. After that when *S* receives a packet *p*, it judges what type of packets that *p* is. If *p* is a data packet, it performs the corresponding statistics for

p. If *p* is an LACK packet issued by an upstream node *U* (the NACK is a reply of an LREQ issued by *U*), *S* creates a new tuple for *U* in its configuration table and establishes an uplink with *U*. The other control packets that *S* receives will be discarded since the MUCOM, MREQ, MEST, LREQ, NACK, RACK, and MACK packets are useless for the sink. When the timer T_P expires, if *S* has received at least one piggybacked packet in T_P, *S* sends an RACK with *Bandwth* = default bandwidth to each of its upstream node *U* to adjust bandwidths for all active node in the WSN. Figure 13 lists the *Sink()* function.

```
Algorithm: Sink()          //performed by the sink of the system
Input: a packet p
Output: a control packet LREQ or RACK

01:  broadcast an LREQ to neighbor nodes;
     //only when system starts up, SenderID = S;
02:  set timer T_P = t units of time;
03:  while (1) {
04:     if (T_P expires) {   //the sink periodically sends an RACK to adjust bandwidth for links
05:        if (received any piggybacked packet in t units of time)
06:           send an RACK with Bandwth = default bandwidth to each of its upstream node
              U;
07:        reset T_P to t units of time; }
08:     else {   //receiving packets
09:        wait for receiving a packet p;
10:        if (p is a data packet)
              //no matter p is an SP_packet, EP_packet or ordinary data packet
11:           perform the corresponding statistics of p;
12:        else if (p is a LACK packet)
13:           create a new tuple for U in the configuration table;
              //tuple(nodeID = S, LinkNodeID = U, LinkType = UP, LinkNun = 1,
              Bandwth = default bandwidth)
14:        else  //p is a control packet of other types
15:           discard the packet; } }
```

Figure 13. The *Sink()* function

The function of *Node()* (see Figure 14) is activated when *N* receives a packet *p* or discovers an event *e*. On discovering *e*, *N* sets a timer to the time period T'_P, calls *eventStart()* to send an SP_packet and establishes alternate links if *N* overflows. When the

timer expires, *N* checks to see whether *e* still exists or not. If yes, it calls *transferData()* (see Figure 15) which generates and sends a data packet with the probability P_{Di} to its downstream node D_i. If *e* disappears, *N* calls *eventStop()* to send an EP_packet and release no-longer required links. *Node()* also checks the type of *p*, and calls *rcvPacket()* (see Figure 16) which contains *rcvDataAtNode()* and *rcvCtrlAtNode()* to process *p*.

```
Algorithm: Node()        //performed by a node N other than the sink
Input: discovering an event e, or receiving a packet p
Output: none

01:  while (1) {
02:     if (e occurs) {
03:        set timer T'P = t' units of time;
04:        eventStart();
           //create a tuple for e in N's configuration table, create an additional downlink for N
           when necessary, and send an SP_packet through the latest established downlink of
           N to the corresponding downstream node
05:           while (1) {
06:              if (T'P expires) {        //N senses its environment periodically
07:                 sense e;
08:                 if (e still exists) {
09:                    transferData();
                       //transfer a data packet generated for e by N to a downstream node Di
                       with probability P^t_Di calculated by using Eq. (4), packets received from
                       upstream nodes are delivered in rcvDataAtNode() function
10:                    reset T'P to t' time units; }
11:                 else {        //e disappears
12:                    eventStop();
                       //delete the corresponding tuple of e from N's configuration table, send
                       an EP_packet through each of the k latest established downlinks to the
                       corresponding downstream nodes where k is calculated by Eq. (1)
13:                    break; } }
14:              else        //T'P does not expire
15:                 rcvPacket();       //discriminate whether p is a control or data packet } }
16:     else     //no event around e occurs
17:        rcvPacket(); }
```

Figure 14. The *Node()* function

Algorithm: transferData() //performed by *N* to deliver a data packet
Input: a data packet *p*
Output: the data packet *p*

01: *send p to the downstream node* D_i *with probability* $P_{Di}^t = \frac{B_{Di}}{\sum_{j=1}^{n} B_{Dj}}$ *where n is the number of downstream nodes of N, and* B_{Di} *is the bandwidth allocated to the link between N and* D_i *by* D_i*;*

Figure 15. The *transferData()* function

Algorithm: rcvPacket() //performed by *N* on receiving a packet
Input: a packet *p*
Output: none;

01: **if** (*p* is a data packet) ***rcvDataAtNode();***
//judge whether *p* is an SP_packet, EP_packet or a data packet and perform the corresponding activities
02: **else if** (*p* is a control packet) ***rcvCtrlAtNode();***
//judge whether *p* is an LREQ, LACK, NACK, a RACK, an MREQ, MACK or an MEST and perform the corresponding activities

Figure 16. The *rcvPacket ()* function

When executing *eventStart()*, *N* creates a tuple *Q* in its configuration table, adds the default data rate DR_{def} to the *Bandwth* field of the tuple corresponding to the latest established downstream node D_n in *N*'s configuration table, and sends an SP_packet containing DR_{def} to D_n. *eventStart()* also calls *linkCreation()* to create alternate links when the value of *BandReq* (see *Eq. (3)*) is larger than 0. On executing *eventStop()*, if there is only one downstream node *D*, *N* deletes the tuple *Q* from its configuration table, subtracts DR_{def} from the *Bandwth* field of *D*'s corresponding tuple in *N*'s configuration table, and sends an EP_packet that contains DR_{def} to *D*. If there are many downstream nodes, *N* deletes the tuple *Q* in *N*'s configuration table, and calls *linkDeletion()* to delete multiple no-longer used links. Figure 17 and 18 respectively lists the *eventStart()* and *eventStop()* functions.

Algorithm: eventStart() //performed by *N* on discovering an event *e*
Input: an event *e*
Output: an SP_packet

01: *create a tuple Q in configuration table for e;*
//tuple(***nodeID*** = N, ***LinkNodeID*** = N, ***LinkType*** = GN, ***LinkNun*** = n+1, ***Bandwth*** = DR_{def}), under the assumption that currently *N* has *n* downlinks

02: **if** (***BandReq*** = $(B_N^t + \sum_{j=1}^{m} B_{Uj}^t) - \sum_{i=1}^{n} B_{Di}^{t+1} > 0$) //see *Eq. (3)*
03: ***linkCreation();*** //create alternative links
04: **else {** //total downlink bandwidth of *N* is sufficient to transmit *N*'s out data
05: *add DR_{def} to the **Bandwth** field of the tuple corresponding to node D_n, which is the latest downstream node linked to N, in N's configuration table;*
//tuple(***nodeID*** = N, ***LinkNodeID*** = D_n, ***LinkType*** = DW, ***LinkNun*** = n, ***Bandwth*** = ***bandwth*** + DR_{def})
06: *send an SP_packet with **Bandwth** = DR_{def} and other source node information to D_n;*
//SP_packet(***eventType*** = SP, ***SrcID*** = N, ***SenderID*** = N, ***Bandwth*** = DR_{def}) **}**

Figure 17. The *eventStart()* function

Algorithm: eventStop() //performed by *N* on the disappearance of event *e*
Input: the disappearance of event *e*
Output: an EP_packet

01: *delete the tuple Q of the disappeared event e from the configuration table;*
//tuple(***nodeID*** = N, ***LinkNodeID*** = N, ***LinkType*** = GN, ***LinkNun*** = --, ***Bandwth*** = DR_{def})
02: **if** (there is only one downstream node *D*) {
03: *subtract DR_{def} from the **Bandwth** field of D's corresponding tuple, i.e., Q, in N's configuration table;*
//tuple(***nodeID*** = N, ***LinkNodeID*** = D, ***LinkType*** = DW, ***LinkNun*** = 1, ***Bandwth*** = ***Bandwth*** - DR_{def})
04: *send an EP_packet with **Bandwth** = DR_{def} to D;*
//EP_packet(***eventType*** = EP, ***SrcID*** = N, ***SenderID*** = N, ***Bandwth*** = DR_{def}) }
05: **else** //there are many downstream nodes
06: ***linkDeletion();***

Figure 18. The *eventStop()* function

linkCreation() broadcasts an MREQ with *BandReq*. *N* on receiving the reply, i.e., MACK, calls *rcvMACK()* to process the MACK packet and establishes an alternate link. The details will be described later. If the *BandReq* is larger than *BandAvail* = default bandwidth, the bandwidth contained in the received MACK, implying more than one downlink are required, *N* rebroadcasts an MREQ.

linkDeletion() deletes *k* no-longer required downlinks, $k \geq 1$, by calculating $X=B_C-B_{Dn}$ where B_C is the *Bandwth* contained in a received EP_packet, and B_{Dn} is the bandwidth allocated to the latest established downlink l_{Dn} between *N* and D_n by D_n. If $X=0$, indicating B_{Dn} will be zero after the bandwidth reduction performed by D_n on receiving the EP_packet, *N* deletes l_{Dn}. If $X<0$, *N* just adjusts the bandwidth of l_{Dn} where $B_{Dn}=B_{Dn}-B_C$. D_n on receiving the EP_packet reduces *X* from the bandwidth of the uplink tuple corresponding to *N* in its configuration table. If $X>0$, representing more than one downlink will be deleted, *N* respectively deletes its downlinks following the reverse order of *LinkNum* from *n* to *2* in its configuration table until $X \leq 0$. After that, for each of the *k* deleted links, e.g., l_m, *N* sends an EP_packet with *Bandwth* = l_m's current allocated bandwidth to the corresponding downstream node D_m, $k \leq m \leq n$, D_m on receiving the packet deletes *N*'s corresponding tuple from its configuration table. For the bandwidth-adjusted link, e.g., l_{k-1}, *N* delivers an EP_packet with the $Bandwth = B_C - \sum_{m=k}^{n} B_{Dm}$. Figure 19 and 20 respectively lists the *linkCreation()* and *linkDeletion()* functions.

```
Algorithm: linkCreation()          //performed by N to create alternate links
Input: none
Output: none

01:  while (1) {
02:     broadcast an MREQ with BandReq;        //see Eq. (3)
03:     while (no MACK has been received from any downstream node)
04:        waiting for a time period;
05:     rcvMACK();     //reply the downstream node D with an MEST packet where the
        first MACK received by N is sent by D, and create a tuple in N's configuration table
        for D
06:     if (BandReq = (BandReq – BandAvail) ≤ 0)        //N has sufficient bandwidth
07:        break; }
```

Figure 19. The *linkCreation()* function

```
Algorithm: linkDeletion()          //performed by N to delete a no-longer required link
Input: none
Output: an EP_packet

01:  while (1) {
02:     if (B_C – B_Dn ≤ 0) {
        //B_C: the Bandwth contained in a received EP_packet, B_Dn: the bandwidth B_n
        allocated to the latest established downlink l_Dn between N and D_n by D_n
03:        send an EP_packet to the downstream node D_n with bandwth = B_C;
           //EP_packet(eventType = EP, SrcID = N, SenderID = N, Bandwth = B_C)
04:        if (B_C – B_Dn == 0)
05:           delete D_n's corresponding tuple from N's configuration table;
              //tuple(nodeID = N, LinkNodeID = D_n, LinkType = DW, LinkNun = n,
              Bandwth = 0)
06:        else //B_C – B_Dn < 0
07:           adjust the tuple of D_n with bandwth = B_Dn – B_C in N's configuration table;
08:        break; }
09:     else {   //B_C > B_Dn, hence, delete multiple links
10:        send an EP_packet to the downstream node D_n with bandwth = B_Dn;
           //EP_packet(eventType = EP, SrcID = N, SenderID = N, Bandwth = B_Dn)
11:        delete D_n's corresponding tuple from N's configuration table;
           //tuple(nodeID = N, LinkNodeID = D_n, LinkType = DW, LinkNun = n, Bandwth
           = 0)
12:        B_C = B_C - B_Dn;        n = n – 1; } }
```

Figure 20. The *linkDeletion()* function

When *N* receives a data packet *p* from an upstream node *U,* the function *rcvDataAtNode()* checks to see whether *p* is a piggybacked packet or not. If not, representing *p* is an ordinary data packet, *N* calls *transferData()* (see Figure 14) to transmit *p*. If yes, it further checks to see whether *N* has a single downlink or multiple downlinks. Note that when *N* receives an SP_packet, if ***BandReq*** > 0, it further calls *linkCreation()* as stated above to create multiple downlinks. Figure 21 lists the *rcvDataAtNode()* function.

```
Algorithm: rcvDataAtNode()   //performed by a node N to process a receiving data packet p
Input: a data packet p issued by an upstream node U
Output: data packet or a piggybacked packet

01:  if (p is a piggybacked packet) {
```

02: **if** (p is an SP_packet) {

03: **if** (***BandReq*** = $(B_N^t + \sum_{j=1}^{m} B_{Uj}^t) - \sum_{i=1}^{n} B_{Di}^{t+1} > 0$) ***linkCreation();*** //see *Eq. (3)*

04: **else if** (there is only one downstream node *D*) {

05: *add the* ***bandwth*** B_u *conveyed on the SP_packet sent by U to the* ***Bandwth*** *field of U's corresponding tuple in the configuration table;*
//tuple(***nodeID*** = N, ***LinkNodeID*** = U, ***LinkType*** = UP, ***LinkNun*** = 1, ***Bandwth*** = ***Bandwth*** + B_u)

06: *add* B_u *to the* ***bandwth*** *field of D's corresponding tuple in N's configuration table;*
//tuple(***nodeID*** = N, ***LinkNodeID*** = D, ***LinkType*** = DW, ***LinkNun*** = 1, ***Bandwth*** = ***Bandwth*** + B_u)

07: *send an SP_packet with* ***Bandwth*** *conveyed on the received SP_packet;*
//SP_packet(***eventType*** = SP, ***SrcID*** = X, ***SenderID*** = N, ***Bandwth*** = B_u) }

08: **else** // there are many downstream nodes D_n

09: *send a new SP_packet with* ***Bandwth*** = B_u *and source node information to* D_n*;*
//SP_packet(***eventType*** = SP, ***SrcID*** = X, ***SenderID*** = N, ***Bandwth*** = D_n) }

10: **else** { //*p* is an EP_packet with ***Bandwth*** = B_d

11: **if** (there is only one downstream node *D*) {

12: *subtract the* ***bandwth*** B_d *from the* ***Bandwth*** *field of U's corresponding tuple in the configuration table;*
//tuple(***nodeID*** = N, ***LinkNodeID*** = U, ***LinkType*** = UP, ***LinkNun*** = 1, ***Bandwth*** = ***Bandwth*** - B_d)

13: *update the* ***bandwth*** *field of D's corresponding tuple in N's configuration table;*
//tuple(***nodeID*** = N, ***LinkNodeID*** = D, ***LinkType*** = DW, ***LinkNun*** = 1, ***Bandwth*** = ***Bandwth*** - B_d)

14: *send an EP_packet with* ***Bandwth*** = B_d *to D;*
//EP_packet(***eventType*** = EP, ***SrcID*** = X, ***SenderID*** = N, ***Bandwth*** = B_d) }

15: **else** // there are many downstream node D_n

16: ***linkDeletion(); } }***

17: **else** //*p* is an ordinary packet

18: ***transferData();*** //forwarding *p*

Figure 21. The *rcvDataAtNode()* function

rcvCtrlAtNode() checks the types of a control packet *c*, calls the *rcvLREQ(), rcvRACK(), rcvMREQ()* and *rcvMACK()* on receiving an LREQ, a RACK, an MREQ and an MACK, respectively. *rcvCtrlAtNode()* creates a tuple for *U* in *N*'s configuration table to establish an uplink between *N* and *U*. When *N* receives an LACK or MEST packet from *Q*. Figure 22 lists

the *rcvCtrlAtNode()* function.

Algorithm: rcvCtrlAtNode() //performed by *N* to process a receiving control packet *c*
Input: a control packet *c* issues by a neighbor node *Q* which may be an upstream node or downstream node
Output: none

01: **if** (*c* is a LREQ packet) ***rcvLREQ();*** //*Q* is a downstream node to create a downlink between *Q* and *N* for the spanning tree
02: **else if** (*c* is a LACK packet)
03: *create a tuple for Q in N's configuration table to establish an uplink between N and Q;*
//*Q* is an upstream node that accepts to establish a link between *Q* and *N* for the spanning tree
04: **else if** (*c* is a NACK packet) *discard c;* //*Q* is an upstream node which refuses to establish a link with *N* for the spanning tree
05: **else if** (*c* is a RACK packet) ***rcvRACK();*** //*Q* is a downstream node which delivers a new bandwidth for the upstream node *N* to allocate bandwidth to *N* or adjust *N*'s bandwidth
06: **else if** (*c* is a MACK packet) ***rcvMACK();*** //*Q* is a downstream node which accepts to establish a link with *N* for an alternative link
07: **else if** (*c* is a MREQ packet) ***rcvMREQ();*** //*Q* is an upstream node to establish an alternative link
08: **else** //receiving a MEST packet from *Q*
09: *create a tuple for Q in N's configuration table to establish an uplink;*
//*Q* is an upstream node which selects *N* as the downstream node to establish an alternate link

Figure 22. The *rcvCtrlAtNode()* function

rcvLREQ() checks to see whether *N* has ever received an LREQ or not. If yes, indicating that *N* has already established a downlink for the spanning tree, no more downlink of *N* is allowed. Hence, *N* discards the packet and replies an NACK. Otherwise, it creates a tuple for the connection in its configuration table, replies an LACK to the sender of the LREQ, e.g., node *Q* which is a downstream node of *N*, and further broadcasts an LREQ to continue establishing a spanning tree. Note that as stated above, only the node with one downlink and without upstream nodes can issue a LREQ.

rcvRACK() (assume the corresponding RACK is issued by *N*'s downstream node *Q*) updates bandwidth field of *Q*'s corresponding tuple in *N*'s configuration table with the

bandwidth conveyed on the received RACK, i.e., adjusting bandwidths for *Q*'s upstream nodes in the rate allocation and adjustment phase. If *N* has upstream nodes, it calculates the bandwidth for each upstream node *U* by using *Eq.(2)*. It further sends an RACK with the newly allocated bandwidth to *U*, and updates bandwidth of *U*'s corresponding tuple in *N*'s configuration table with the bandwidth conveyed on the RACK sent.

rcvMREQ() checks to see whether the link between *N* and *Q* exists or not. Here, *Q* is one of *N*'s upstream nodes. If yes, *N* discards the received packet. Otherwise, *rcvMREQ()* replies an MACK with ***BandAvail*** = default bandwidth.

If *N* has ever received current MREQ's corresponding MACK from other nodes, *rcvMACK()* discards current receiving MACK. It allocates bandwidth with the minimum value of ***BandReq*** and ***BandAvail*** to *D,* where the first corresponding MACK packet with ***BandAvail*** received by *N* is sent by *D*. *N* further creates a tuple for *D* in the configuration table to establish a downlink, and replies *D* with an MEST. Figures 23~26, respectively, list the functions of *rcvLREQ(), rcvRACK(), rcvMREQ()* and *rcvMACK()*.

```
Algorithm: rcvLREQ()          //performed by N to process a receiving LREQ packet
Input: a control packet LREQ issued by a downstream node Q
Output: a control packet LACK or NACK

01:  if (N has ever received an LREQ from a downstream node Q)
02:       {discard the LREQ;    reply an NACK; }
          //in link establishment phase, a node can only establish a downlink
03:  else {
04:       create a tuple for Q in N's configuration table;
          //tuple(nodeID = N, LinkNodeID = D, LinkType = DW, LinkNun = 1, Bandwth =
          default bandwidth)
05:       reply Q with an LACK;    broadcast an LREQ; }
```

Figure 23. The *rcvLREQ()* function

Algorithm: rcvRACK() //performed by *N* for bandwidth adjustment
Input: a control packet *RACK* issued by a downstream node *D*
Output: a *RACK*

01: *replace bandwidth field of D's corresponding tuple in N's configuration table with the* ***bandwth*** *conveyed on the RACK;*
02: **if** (*N* has upstream nodes) {
03: *calculate bandwidth for each upstream node* U_i *for time* $t+1$ *by invoking Eq.(2),* $i = 1, 2, ..., m$*;* //assume *N* currently has *m* uplinks
04: *send an RACK to* U_i *telling* U_i *the newly allocated bandwidth* B_{Ui}*;*
05: *update bandwidth of* U_i*'s corresponding tuple in N's configuration table with the bandwidth* B_{Ui}*; }*
//tuple(***nodeID*** = N, ***LinkNodeID*** = U, ***LinkType*** = UP, ***LinkNun*** = 1~n, ***Bandwth*** = B_{Ui})

Figure 24. The *rcvRACK()* function

Algorithm: rcvMREQ() //performed by *N* to establish an alternate uplink
Input: a control packet *MREQ* issued by an upstream node *U*
Output: a control packet *MACK*

01: **if** (MREQ is sent by *U* and there is an existing link between *N* and *U*)
02: *discard the MREQ;*
03: **else**
04: *reply U with an MACK with* ***BandAvail*** *= default bandwidth;*

Figure 25. The *rcvMREQ()* function

Algorithm: rcvMACK() //performed by *N* on receiving an MACK from a downstream node *D*

Input: a control packet *MACK* issued by a downlink *D*

Output: a control packet *MEST* or none

01: **if** (*N* has ever received current MREQ's corresponding MACK from other nodes)
discard the MACK; //one MREQ establishes only one alternate link

02: **else** {

03: **if** (the ***RecverID*** from received MACK == the ***NodeID*** in *N*'s configuration table) {

04: *allocate bandwidth with the min(**BandReq, BandAvail**) to D, where the first MACK packet with **BandAvail** , which is the response of the recently sent MREQ, received by N is sent by D;*

05: *create a tuple for D in the configuration table to establish a downlink;*
//tuple(***nodeID*** = N, ***LinkNodeID*** = D, ***LinkType*** = DW, ***LinkNun*** = n, ***Bandwth*** = min(***BandReq, BandAvail***))

06: *reply D with an MEST with **Bandwth*** = min(***BandReq, BandAvail***)*;*}

07: **else** *discard the MACK;* //the link between *N* and *D* already exists}

Figure 26. The *rcvMACK()* function

IV. Simulation Results and Discussion

We use ns-2 [16] as the simulation tool to evaluate the MUCOM, and compare it with HCCP [1] and the method without employing congestion control (WECC for short). The default parameters for all experiments are listed in Table 5. The sink is placed at the center (50, 50) of the 100m x 100m field. 49 sensor nodes were randomly deployed in the field for sensing their surrounding environments and relaying packets. The sink only collects data packets sent by sensor nodes.

Table 5. Parameters of the experimental environment

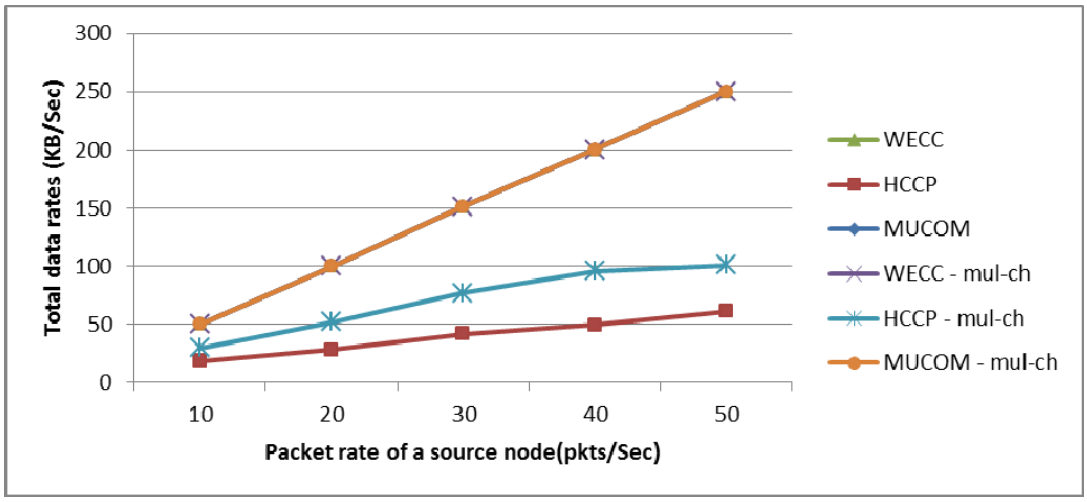
Parameter	Value
Number of sink node	1
Number of sensor nodes	49
Experimental field	100 x 100 m^2
MAC layer protocol	IEEE 802.11
Max bandwidth of a link	250 Kbps (=31.25 KB/sec)
Radio transmission range of a node	20 m
Experimental duration	100 sec
Number of simulations for each experiment	50
Packet rate of a source node	10 pkts/sec
Packet size	1 KB/pkt
Number of events occurs for each experiment	5
Event lasting time	25 sec

In this study, five experiments were performed. The first evaluated throughputs defined as the cumulative data size received per second by the sink, end-to-end delays defined as the time period from when a packet is sent by its source node to the time point when the sink receives the packet, and packet drop rates defined as (number of packets sent by all source nodes – number of packet received by the sink) over number of packets sent by all source nodes given different packet rates. The second, third, fourth and fifth experiments redid the first experiment, respectively, on different packet sizes, numbers of events generated, event-lasting times, and node densities. The default parameters for each experiment may be changed when necessary.

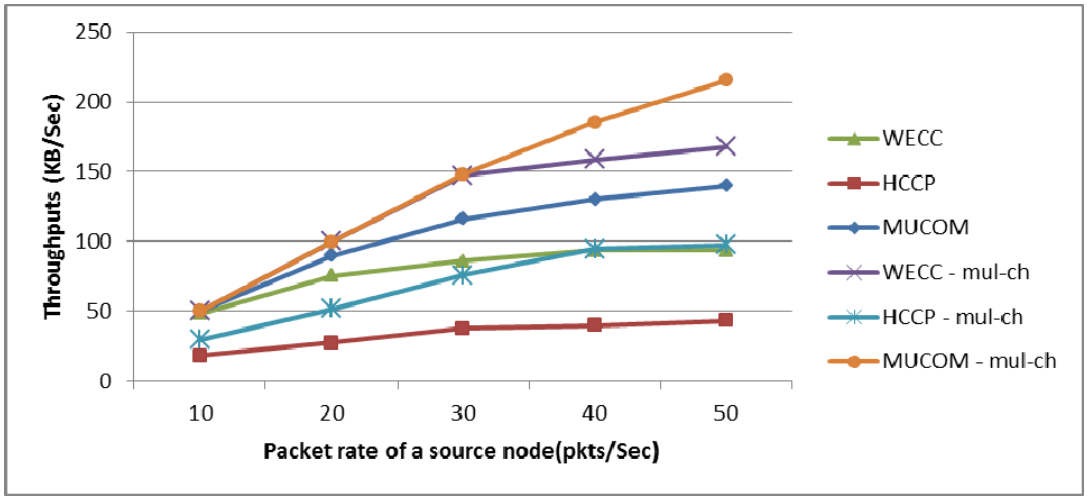
4.1. Different Data Rates

In the first experiment, each active source node sends packets to the sink on different

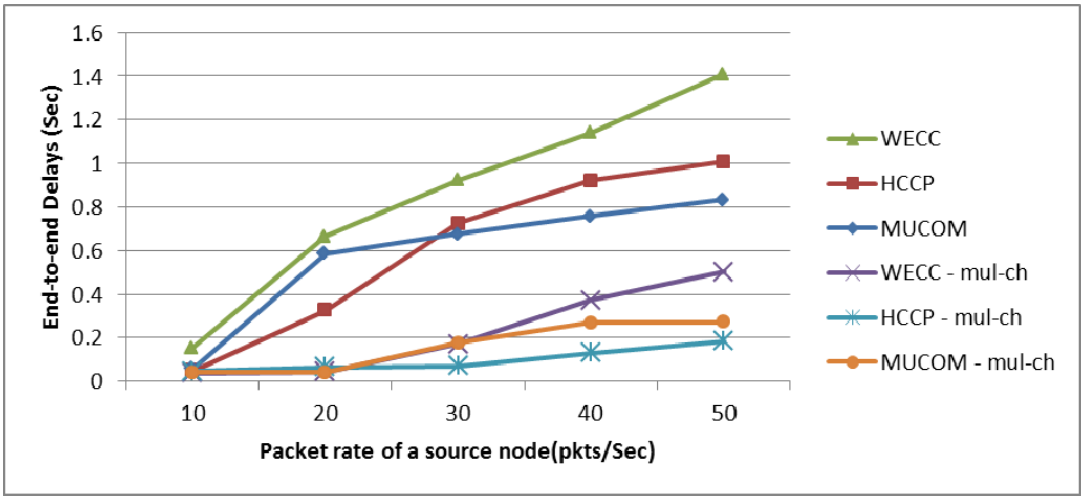
packet rates ranging between 10pkts/sec (i.e., 10KB/sec = 10pkts/sec x 1KB/pkt) and 50pkts/sec (i.e., 50KB/sec = 50pkts/sec x 1KB/pkt), instead of 10pkts/sec listed in Table 5. Some of the total data rates issued by the five source nodes are higher than a link's bandwidth. The data rates of the tested schemes are shown in Figure 27a, from which we can see that the HCCP's are not linear. Figures 27b, 27c and 27d respectively show the throughputs, end-to-end delays, and packet drop rates measured at the sink. Here, mul-ch standing for multichannel represents that all the neighbor nodes of the sink are given different channels to avoid channel contention when packets are sent between these nodes and the sink. We call this a multi-channel environment, and call these nodes the sink-neighbor nodes. The symbols of WECC, HCCP, and MUCOM with mul-ch mean the sink-neighbor nodes use the same channel so that before sending packets to the sink, they need to contend the channel. We call this a single-channel environment in which packet collision may occur.



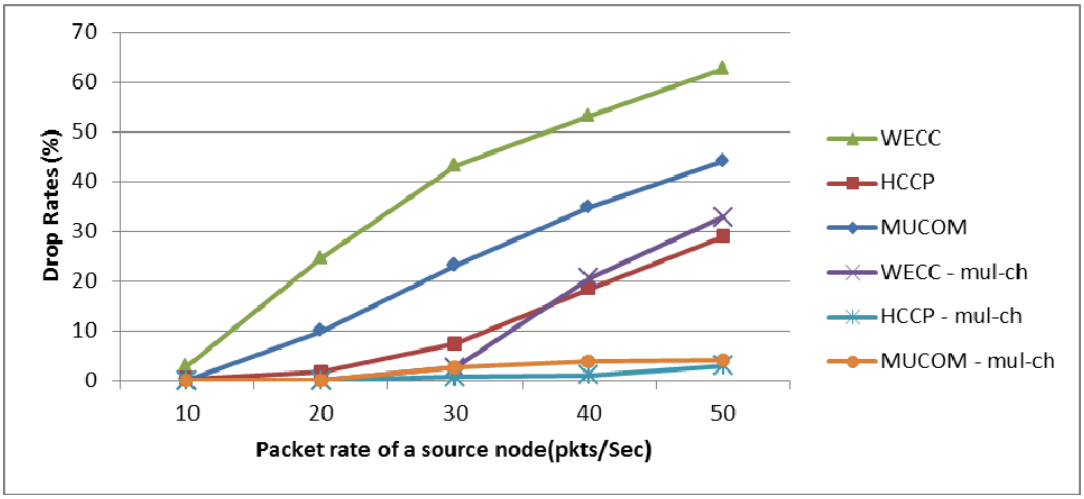
(a) Total packet rates sent to the sink on packet size = 1KB



(b) Throughputs at the sink on packet size = 1KB



(c) Average end-to-end delays on packet size = 1KB



(d) Average packet drop rates on packet size = 1KB

Figure 27. Experimental results of the tested algorithms on different packet rates when packet size = 1KB

In the given experimental topology, there are a total of seven sink-neighbor nodes. Therefore, theoretically, the data rate can be up to 218.75KB/sec (=31.25KB/sec x 7). When the data rates increase, many more packets are sent per second so that the throughputs of the tested schemes as shown in Figure 27b are higher. Due to employing multipath data transfer and rate-based congestion control throughout the WSN, the MUCOM and MUCOM-mul-ch outperform the other two, no matter in the single-channel or the multi-channel environment. In the latter environment, owing to no channel contention among the sink-neighbor nodes, the MUCOM-mul-ch's throughputs are higher than those of the other schemes, in the former environment, showing that channel contention among these nodes is serious. The HCCP's throughputs are the lowest both in the single-channel and multi-channel environments since it

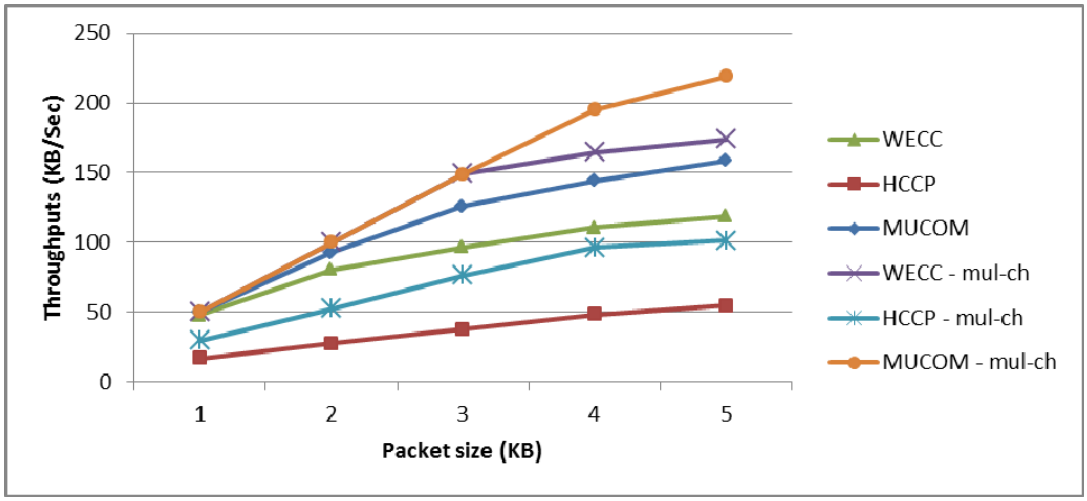
over-pressures the data rates, causing less data packets being transmitted to the sink, even though the HCCP has lower end-to-end delays and drop rates than the WECC and MUCOM have (see Figures 27c and 27d). When the data rates exceed 31.25 KB/sec, i.e., the data generated per second by a source node exceeds the bandwidth of a link, the throughputs of the MUCOM-mul-ch increase continuously since the employment of additional paths widens the data delivery bandwidth.

In this experiment, the end-to-end delay of a packet, e.g., packet A, includes the time of retransmitting A once A is dropped. In the experimental field, the average number of hops of a path from the source node to the sink is about 3.51, i.e., there are 3.51 hops that a packet has to travel from its source node to the sink. In Figure 27c, when the data rates are higher, owing to higher probability of network congestion, the delays of the tested schemes increase. Also, the MUCOM and MUCOM-mul-ch need some time to establish an alternate path for a congested node. This also prolongs their delays. Due to few channel contention, the delays of a tested scheme in the multi-channel environment are much shorter than those of itself in the single-channel environment, showing that channel contention seriously lengthens packet delivery delays, particular for the nodes in the area near the sink. The HCCP's delays are lower than those of the MUCOM and the WECC when the data rates are less than 30 KB/sec because when detecting packet congestion, it suppresses its data rate to reduce the data drop rates. When the data rates exceed 30 KB/sec, the delays of the HCCP increases sharply and are longer than those of the MUCOM since in the single-channel environment when data rates are over the bandwidth of a link, the HCCP does not establish alternate paths to help the delivery of packets. Although the delays are shorter than those of the HCCP, the MUCOM and MUCOM-mul-ch keep their data rates and use multiple links to improve system performance. That is why their delays are shorter than those of the WECC.

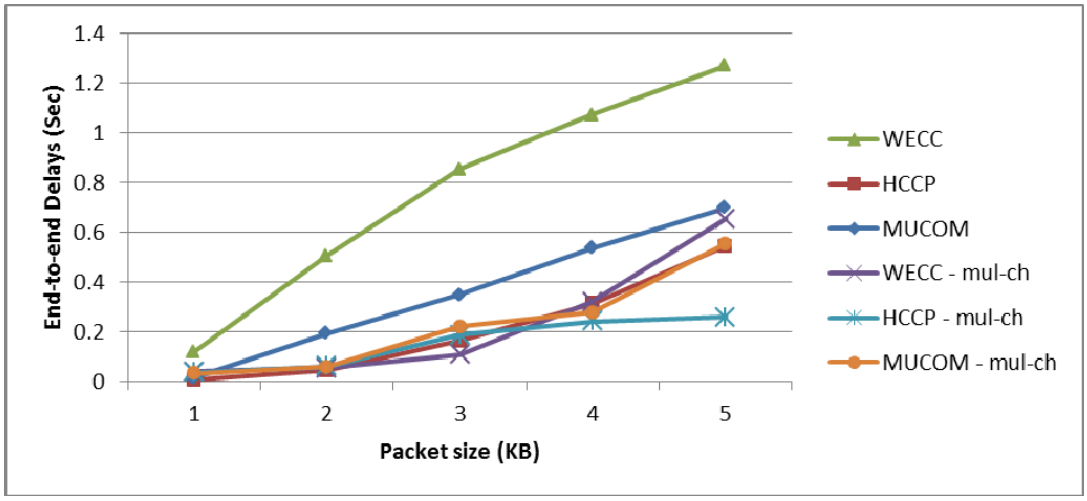
In Figure 27d, due to higher probabilities of packet collision and congestion, larger data rates result in higher packet drop rates. The drop rates of the WECC-mul-ch increase sharply when the data rate exceeds 30KB/sec, although the multi-channel environment relieves the channel contention among the sink-neighbor nodes, indicating that when data rates of a source node exceed the bandwidth of a link, a method to migrate channel congestion is required. This is also one of the motivations of this study. The MUCOM's drop rates are lower than those of the WECC, showing that the use of multiple paths can effectively shorten end-to-end delay. Besides, the multi-channel environment can further improve a scheme's the throughputs, end-to-end delays, and drop rates.

4.2. Different Packet Sizes

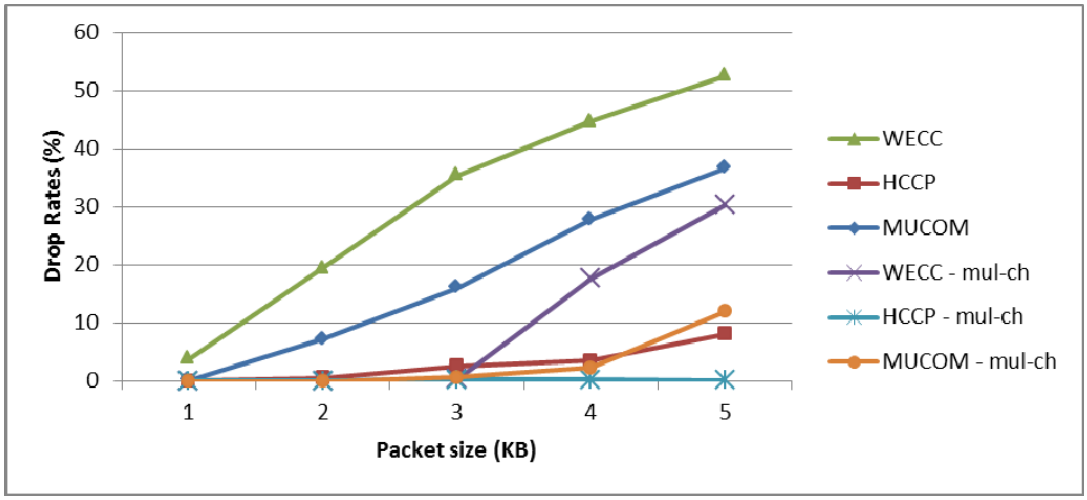
In the second experiment, the packet sizes range between 1KB and 5KB instead of 1KB listed in Table 5. The packet rate of an active source node is constantly 10pkts/sec. Hence, the data rates are between 10KB/sec (= 10pkts/sec x 1KB/pkt) and 50KB (= 10pkts/sec x 5KB/pkt). The total data rates of the tested schemes are individually the same as those shown in Figure 27a. Figures 28a, 28b and 28c respectively show the throughputs, end-to-end delays, and packet drop rates of this experiment.



(a) Throughputs at the sink on packet rate = 10pkts/sec



(b) Average end-to-end delays on packet rate = 10pkts/sec



(c) Average packet drop rates on packet rate = 10pkts/sec

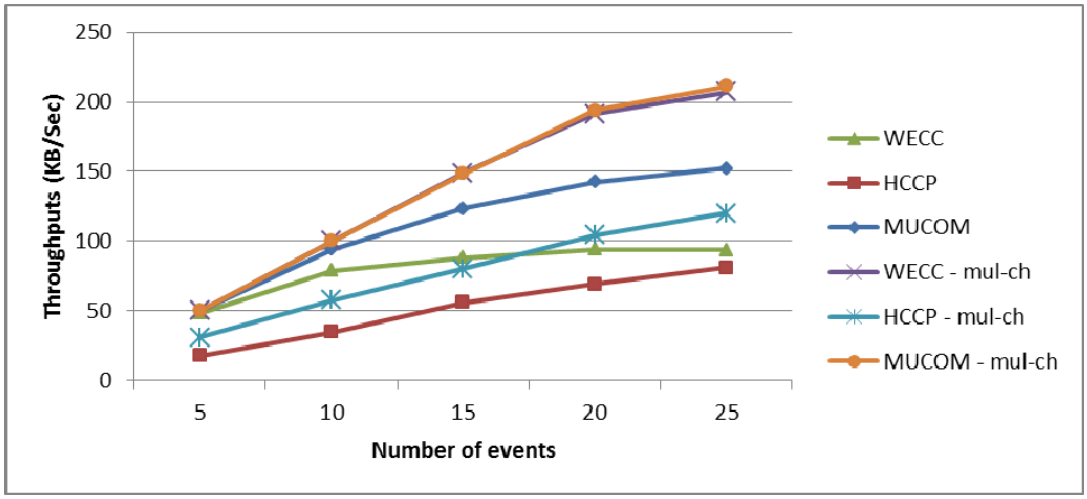
Figure 28. Experimental results of the tested algorithms on different packet sizes

When the packet sizes increase from 1KB to 5KB, the trends of the throughputs of the tested schemes in both environments (single-channel and multi-channel) are similar to those shown in the first experiment. Even though the indexes of the X-axises of Figure 27 and 28 are different, the total data rates of the five event source nodes on the i-th index are the same, i=1, 2… 5. For example, in Figure 27, the first index of the X-axis is 10 KB/sec (=10pkts/sec x 1KB/pkt), and the total data rate is 50 KB/sec (=10KB/sec x 5 events). In Figure 28, the first index of the X-axis is 1KB, the total data rate is also 50KB/sec (= 1KB/pkt x 10pkts/sec x 5 events). In the two figures, the total data rates of the 2nd to the 5th indexes are respectively 100, 150, 200 and 250 KB/sec. Since we fix packet sizes, the packet rates of the first experiment are 20, 30, 40, and 50pkts/sec, instead of 10pkts/sec. Due to delivering fewer numbers of packets, the probabilities of packet collision and channel contention of this experiment is mitigated, resulting in higher throughputs, and lower delays and drop rates. We now conclude that in a WSN when data rates are the same, transmitting fewer packets will result in better performance than that of delivering many more packets.

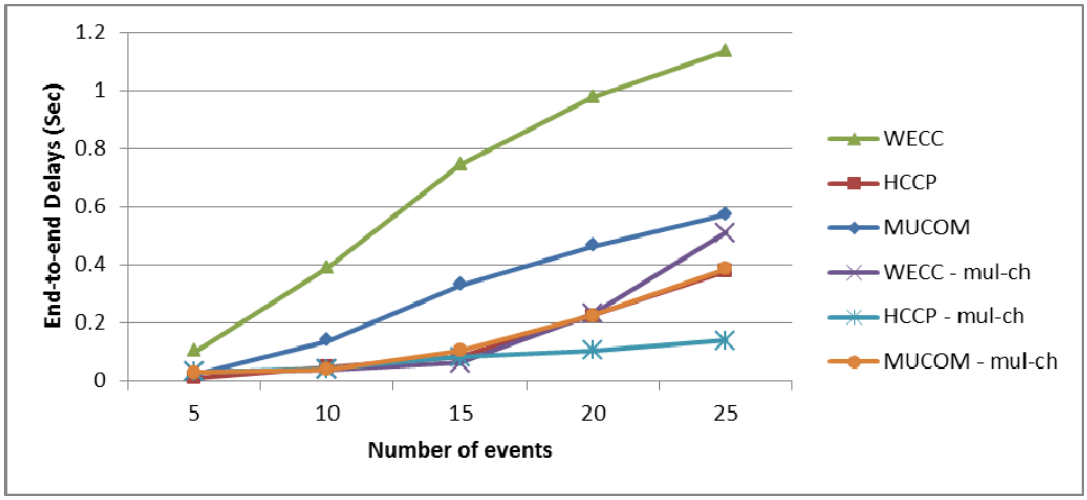
4.3. Different Number of Events

In the third experiment, the numbers of events are 5, 10, 15, 20, and 25 rather than 5 listed in Table 5. In fact, the total data size generated by all active source nodes for the i-th index of the X-axis of Figure 29, link that described in the second experiment, is the same as that generated for the i-th index of the X-axis in Figure 27, i=1, 2,…, 5, i.e., the total data rates sent to the sink by the tested schemes are also individually the same as those shown in Figure 27a. Events are randomly generated in the field. Figures 29a, 29b and 29c respectively

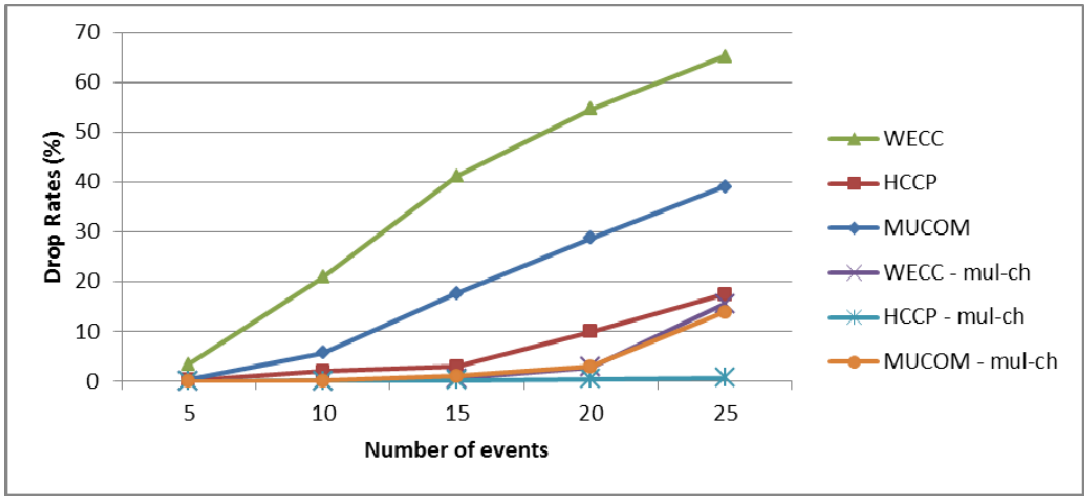
show the throughputs measured, end-to-end delays, and packet drop rates.



(a) Throughputs at the sink



(b) Average end-to-end delays



(c) Average packet drop rates

Figure 29. Experimental results of the tested algorithms on different number of events

Since the data rate generated by a source node, i.e., 10KB/sec (= 10pkts/sec x 1KB/pkt), does not exceed the bandwidth of a link, i.e., 31.25KB/sec, and the events may disperse in the field instead of only occurring on five fix nodes, the probabilities of bandwidth overflow on a link and channel contention are lower, causing higher throughputs and lower delays (see Figures 29a and 29b) compared with those of the first experiment (see Figures 27b and 27c).

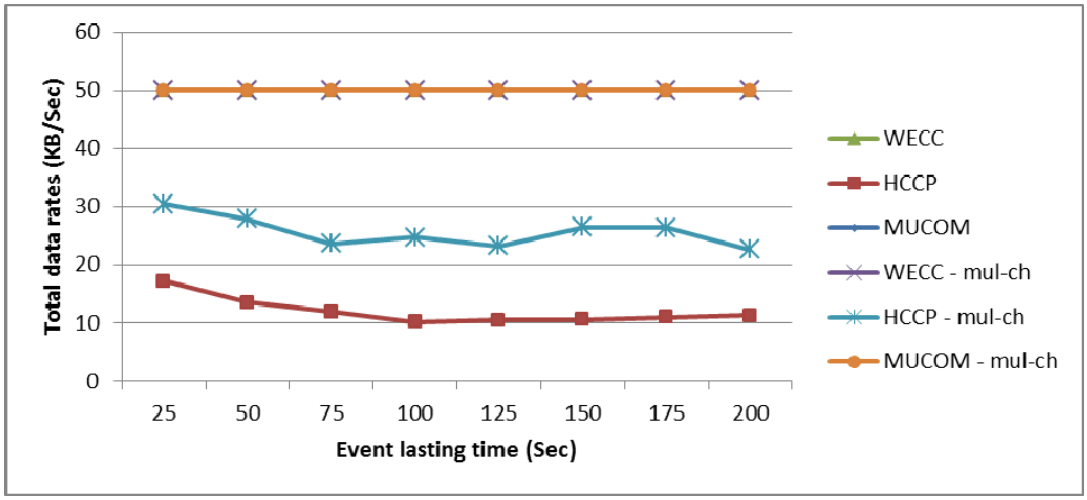
When the numbers of events increase, the probability of packet collision is also higher. In Figure 29a, the throughputs of the WECC-mul-ch are similar to those of the MUCOM-mul-ch even when event numbers = 20 or 25 (please also refer to Figures 27b and 28a). This is because events disperse in the field. The probability that the data rate flowing through a link is higher than the link's bandwidth is lower. This also a key reason why the throughputs of the tested schemes are generally higher than those of themselves in the first experiment.

As with the previous experiments, the delays and drop rates increase when numbers of events are higher. The reason is the increased numbers of packets causes the higher probabilities of packet collision, the cumulated data sizes own to higher bandwidth occupation, and the interference between nodes resulting to channel contention. Since the number of channel contention is generally reduced in the areas of sink-neighbor nodes, the tested schemes' delays and drop rates are lower in the multi-channel environment except WECC-mul-ch on event number = 20 and 25, due to the serious packet collisions. The bandwidth occupation probability is also lower than those of the previous experiment due to the data rates cannot exceed the bandwidth of a link, the channel contention among the nodes

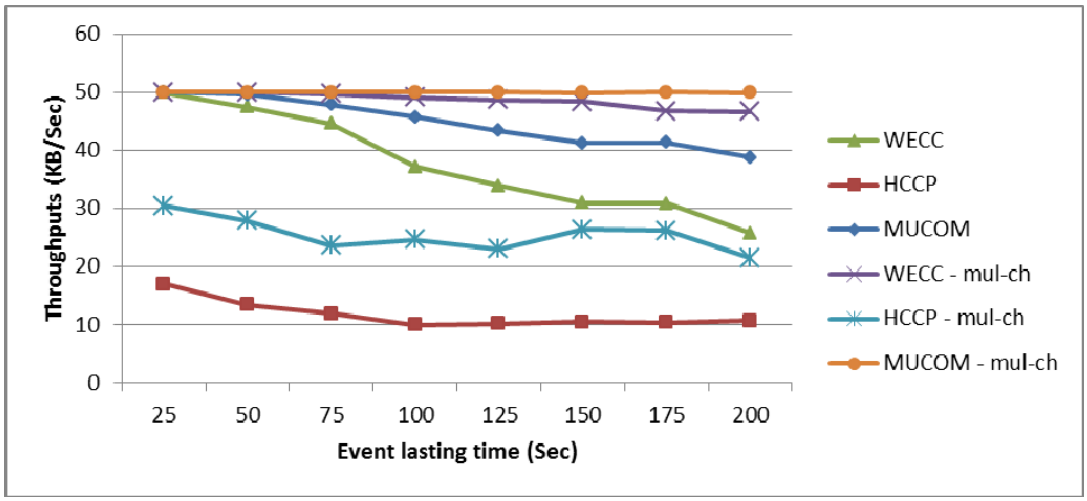
other than the sink-neighbor nodes and the packet collision is the main reason resulting in longer delays and higher drop rates in the multi-channel environment. By employing their congestion control scheme, the MUCOM and HCCP gets lower delays and drop rates than the WECC.

4.4. Different Event-lasting Times

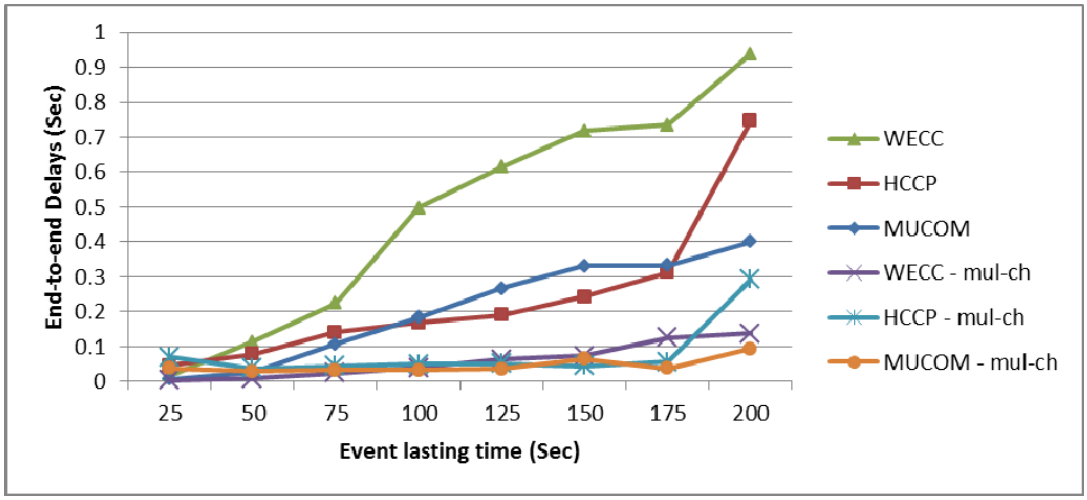
In the fourth experiment, the event-lasting times range between 25 and 200 sec instead of 25 sec listed in Table 5. The start time of an event is random. Figures 30a, 30b, 30c and 30d respectively show the total data rates generated by all active source nodes, their throughputs, end-to-end delays, and packet drop rates.



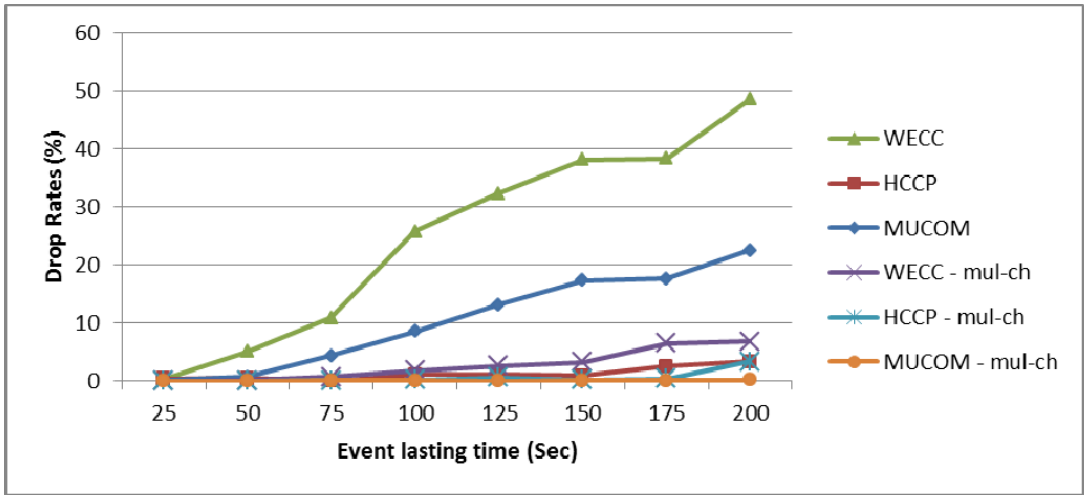
(a) Total data rates



(b) Throughputs at the sink



(c) Average end-to-end delays



(d) Average packet drop rates

Figure 30. Experimental results of the tested algorithms on different event-lasting times

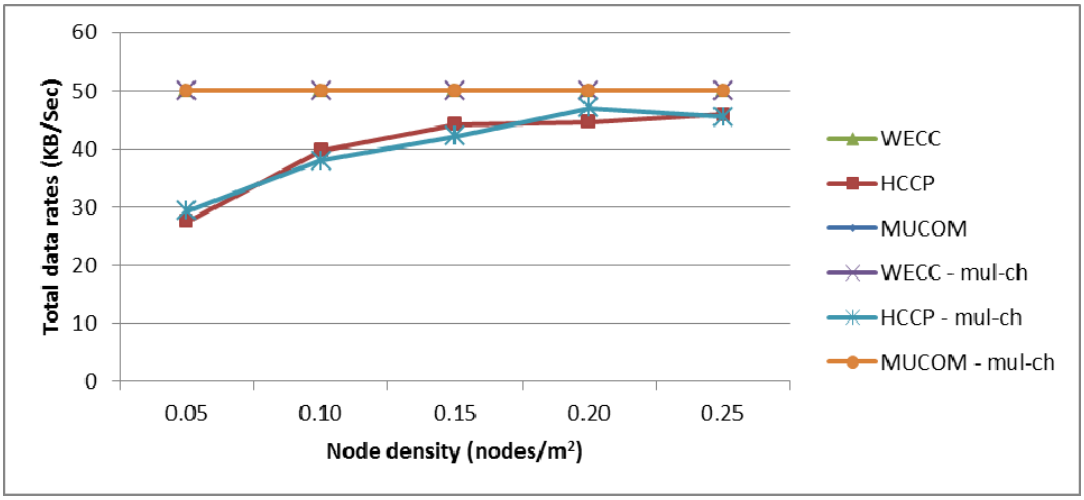
Generally, a longer event-lasting time represents that more packets are sent by a source node. Figure 30a shows that the total data rates of the WECC and MUCOM are a constant 50KB/sec (1KB/pkt x 10pkt/sec x 5events). When the event-lasting times increase, due to the increase of the numbers of packets, the packet collision and contention are worse, causing lower throughputs and higher delays and drop rates. The throughputs of the HCCP and HCCP-mul-ch follow their data rates (please compare Figures 30a and 30b). The phenomena of channel congestion and packet collision among the sink-neighbor nodes appear again between the two lines of the MUCOM and MUCOM-mul-ch, and between the two lines of the WECC and the WECC-mul-ch. Similar to the previous three experiments, the tested schemes' throughputs in the multi-channel environment are better than those of themselves in the single-channel environment. Due to over-suppressing its data rates on detecting packet

congestion, the HCCP's throughputs are the lowest both in the two environments, although its delays and drop rates, like those in pervious experiments, are lower than those of the other test schemes (see Figures 30c and 30d).

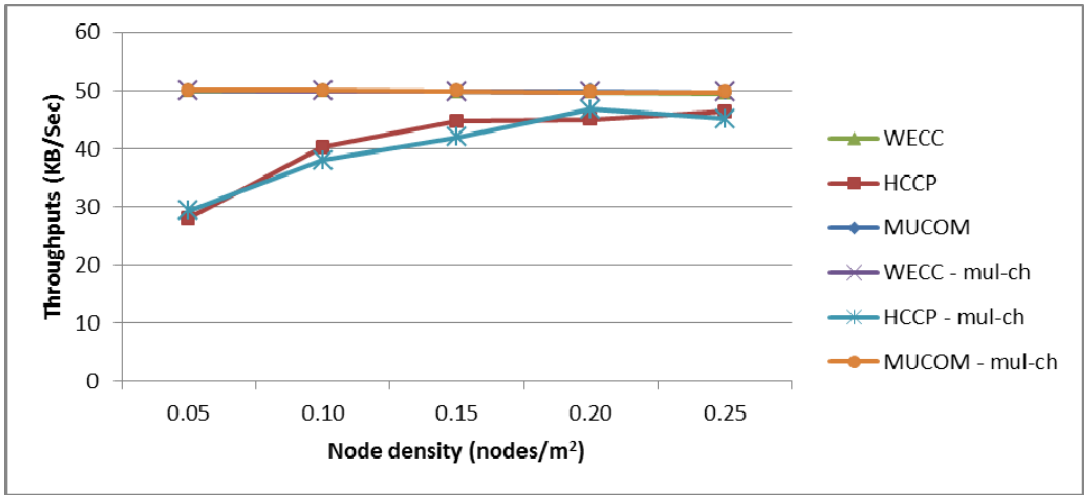
In this experiment, the longer event-lasting time causes higher probabilities of packet collision, and the retransmission of dropped packets lengthens the delays, resulting in longer delays and higher drop rates. As with the third experiment, the data rates generated by each source nodes does not exceed the bandwidth of a link so that the probability of packet congestion is lower than those in the previous three experiments (comparing 27c, 28b, 29b, and 30c, and comparing 27d, 28c, 29c and 30d). The tested schemes in the multi-channel environment have shorter delays and lower drop rates than they in single-channel environment have. The reason is mentioned above. The HCCP's delays increase sharply when the event-lasting time is 200 second because the HCCP's congestion control cannot efficiently reduce packet collision in such a long event-lasting time. The MUCOM's (the MUCOM-mul-ch's) delays and drop rates are lower than those of the WECC (WECC-mul-ch) due to using the multiple path data transfer and rate-based bandwidth control.

4.5. Different Node Densities

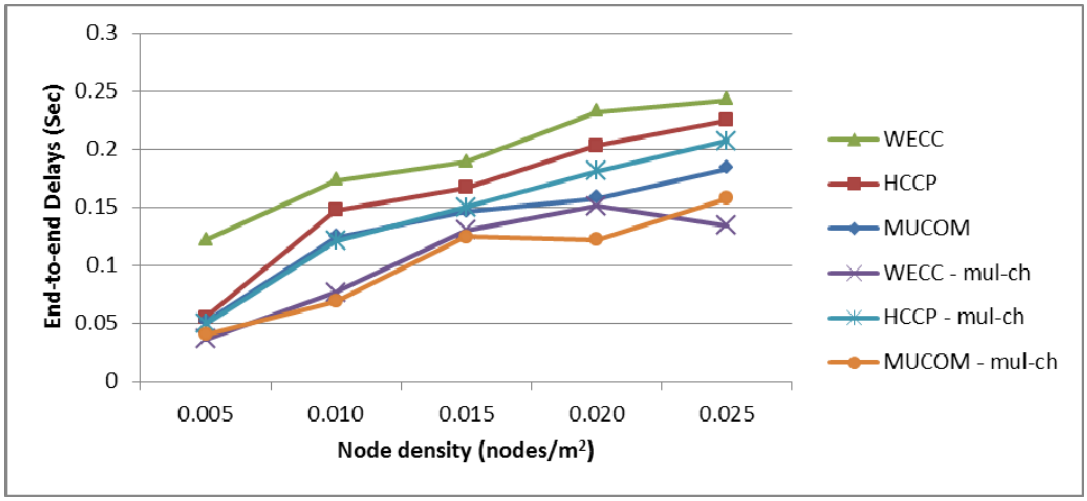
In the fifth experiment, the number of nodes distributed the field range between 50 and 250 instead of 50 listed in Table 5, i.e., the node densities of the field range from 0.005nodes/m^2 to 0.025nodes/m^2 rather than 0.005nodes/m^2. The role and the position of the sink are the same as those of the previous experiments. The packet rate is 10pkts/sec, packet size is 1KB/pkt, 5 events occur in each simulation and each event lasts 25 seconds. In the multi-channel environment, the 11 channels of the WiFi are employed, 10 for the sink-neighbor nodes and one for the other nodes. Figures 31a, 31b, 31c and 31d respectively show the total data rates generated by all active source nodes, their throughputs, end-to-end delays, and packet drop rates.



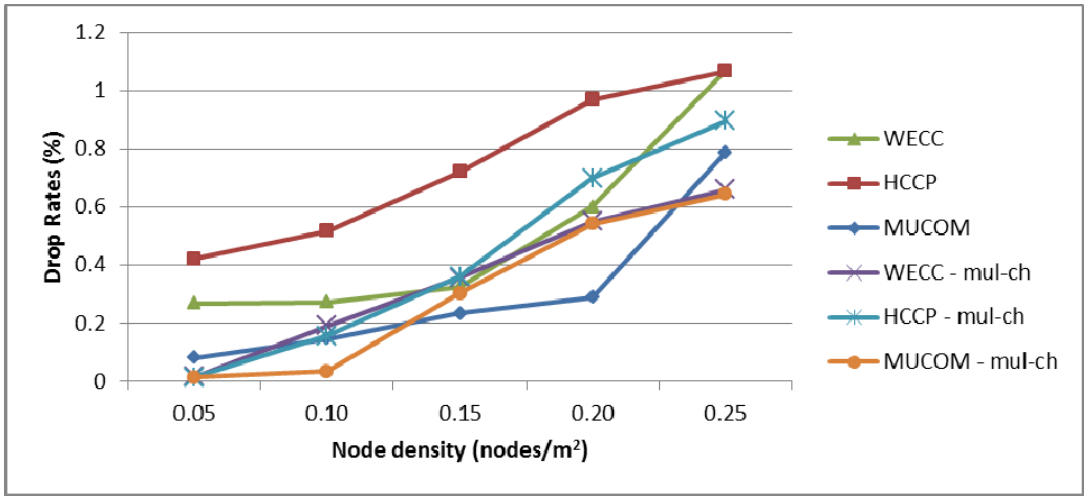
(a) Total data rates



(b) Throughputs at the sink



(c) Average end-to-end delays



(d) Average packet drop rates

Figure 31. Experimental results of the tested algorithms on different node densities

In Figure 31a, the total data rates of the MUCOM and WECC are 50KB/sec (=10pkts/sec x 1KB/pkt x 5events), and the HCCP and HCCP-mul-ch are lower than 50KB/sec due to its data rate suppression. When the node densities increase, the data rates of HCCP and HCCP-mul-ch are higher because the increase of node densities creates many more available paths that the HCCP can choose for a source node, when congested, there is no difficulty for it to choose another path, resulting in less data-rate suppression.

In the multi-channel environment, when the node density exceeds 0.01, the number of sink-neighbor nodes is higher than the number of channels assigned to those nodes so that some of the sink-neighbor nodes are assigned the same channel. Because of less channel contention, packet collision, and bandwidth congestion, the throughputs of the tested schemes are almost the same as their total data rates, implying their drop rates are low (less than 1.2%) (see Figures 31b and 31d). The multi-channel effect in this experiment is insignificant compare to those in the previous experiments.

Generally, higher node density implies that many more nodes will be contained in a routing path, resulting in larger hop counts and lower energy consumptions. In Figure 31c, when node densities increase, due to the increase of the numbers of hops of a routing path, the delays are longer. The tested schemes' delays in the multi-channel environment are shorter than those of themselves in the single-channel environment, although some of the sink-neighbor nodes are allocated the same channel. The performance of MUCOM and MUCOM-mul-ch are also better than those of the other tested schemes.

V. Conclusions and Future Work

In this paper, we propose a MUCOM which achieves fairness of packet delivery in a WSN based on dynamic rate allocation, which at first establishes a spanning tree to connect all nodes for initial packet delivery. When a node's traffic overflows, the MUCOM establishes an alternate path to deliver data packets. In other words, it is a multi-path routing environment. To mitigate congestion for a node, the sink adjusts allocated bandwidths for its upstream nodes which will in turn adjust bandwidth for each of it upstream nodes. The process repeats until no more upstream nodes exist.

Experimental results show that the performance of the MUCOM outperforms the other two schemes. Generally, when the data rates of an event, event packet sizes, event numbers, event-lasting times and the node densities increase, the performance of a WSN in delays and drop rates decrease due to heavier packet collision, bandwidth congestion and channel contention. In the single-channel environment, the channel contention among the sink-neighbor nodes is serious, the multi-channel environment can improve a scheme's throughputs, end-to-end delays, and drop rates. Experimental results also shows that when data rates of a source node exceed the bandwidth of a link, a method to migrate bandwidth congestion is required, and when data rates are the same, due to less time of packet collision and contention, the performance of transmitting fewer packets will be better than the that of delivering many more packets. By employing the multipath data transfer and rate-based congestion control, the MUCOM can effectively improve a WSN's throughputs, end-to-end delays and reduce packet loss rates.

In the future we will derive the reliability model and behavior model for the MUCOM in the WSN environment so that users can predict the system's reliability and behavior before using it. We will also develop an optimized spanning tree construction algorithm to further improve the performance of a WSN. These constitute our future studies.

References

[1] J.P Sheu, L.J. Chang and W.K. Hu, "Hybrid congestion control protocol in wireless sensor networks," *Journal of Information Science and Engineering*, vol. 25, 2009, pp. 1103-1119.

[2] F.B. Hussain, G. Seckin and Y. Cebi, "Many-to-one congestion control scheme for densely populated WSNs," *IEEE/IFIP International Conference in Central Asia*, 2007, pp. 1-6.

[3] C. Wang, B. Li, K. Sohraby, M. Daneshmand and Y. Hu, "Upstream congestion control in wireless sensor networks through cross-layer optimization," *IEEE Journal on Selected Areas in Communications,* vol. 25, 2007, pp. 786-795.

[4] V.P. Munishwar, S.S. Tilak and N.B. Abu-Ghazaleh, Congestion and Flow Control in Wireless Sensor Networks, *Guide to Wireless Sensor Networks*, Springer London, 2009.

[5] X. Yin, X. Zhou, R. Huang, Y, Fang and S. Li, "A fairness-aware congestion control scheme in wireless sensor networks,"*, IEEE Transactions on Vehicular Technology*, vol. 58, issue.9, 2009, pp. 5225-5234.

[6] M.M. Monowar, M.O. Rahman and C.S Hong, "Multipath congestion control for heterogeneous traffic in wireless sensor network," *International Conference on Advanced Communication Technology, 2008*. pp. 1711-1715.

[7] H. Morino, H. Kawamura, M. Inoue and T. Sanefuji, "Load-balanced multipath routing for wireless mesh networks: A step by step rate control approach," *International Symposium on Autonomous Decentralized Systems, 2009*. pp. 1-6.

[8] H.W. Oh, J.H. Jang, K.D. Moon, S.C. Park, E.S. Lee, and S.H. Kim, "An explicit disjoint multipath algorithm for Cost efficiency in wireless sensor networks," *IEEE International Symposium on Personal Indoor and Mobile Radio Communications,* 2010, pp. 1899-1904.

[9] N. Stephan, S. Varakliotis, and P. Kirstein, "Transport layer multipath on wireless sensor network backhaul links," *International Conference on Sensor Technologies and Applications,* 2009, pp. 469-472.

[10] W.J. Lou, "An efficient N-to-1 multipath routing protocol in wireless sensor networks," *International Conference on Mobile Adhoc and Sensor Systems*, 2005, pp. 665-672.

[11] H. Fariborzi and M. Moghavvemi, "EAMTR: energy aware multi-tree routing for wireless sensor networks," *IET Communications,* vol. 3, no. 5, 2009, pp. 733-739.

[12] Y. Sankarasubramaniam, Ö.B. Akan and I.F. Akyildiz, “ESRT: event-to-sink reliable transport in wireless sensor networks,” *the ACM international symposium on Mobile ad hoc networking & computing*, 2003, pp. 177-188.

[13] C.Y. Wan, S.B. Eisenman and A.T. Campbell, “CODA: congestion detection and avoidance in sensor networks,” *the international conference on Embedded Networked Sensor Systems*, 2003, pp. 266-279.

[14] Y. Xiao, “Concatenation and piggyback mechanisms for the IEEE 802.11 MAC,” *Wireless Communications and Networking Conference,* 2004, pp. 1642-1647.

[15] B.H. Walke, S. Mangold and L. Berlemann, *IEEE 802 Wireless Systems: Protocols, Multi-Hop Mesh/Relaying, Performance and Spectrum Coexistence*. Wiley, 2007.

[16] *ns-2*, http://www.isi.edu/nsnam/ns/

Appendix

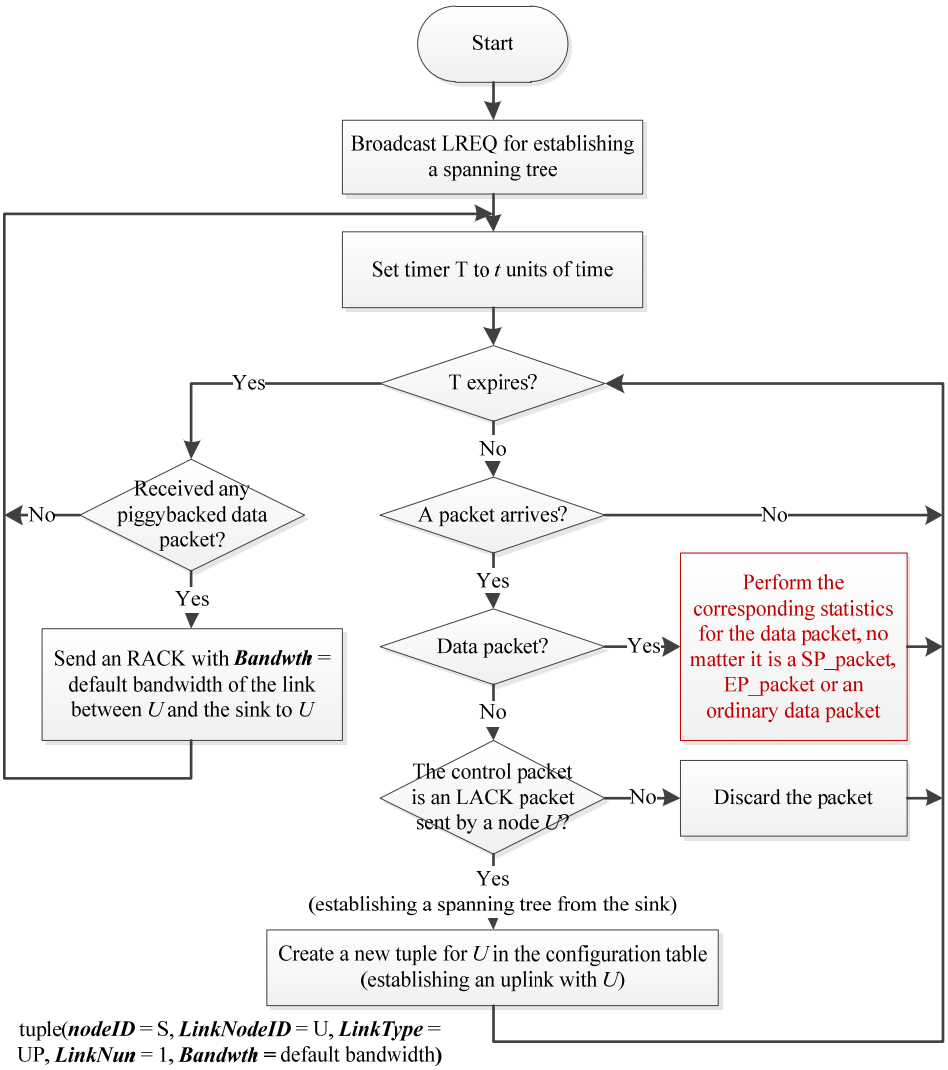


Figure A1. Flow chart for the operations performed by the sink *S*

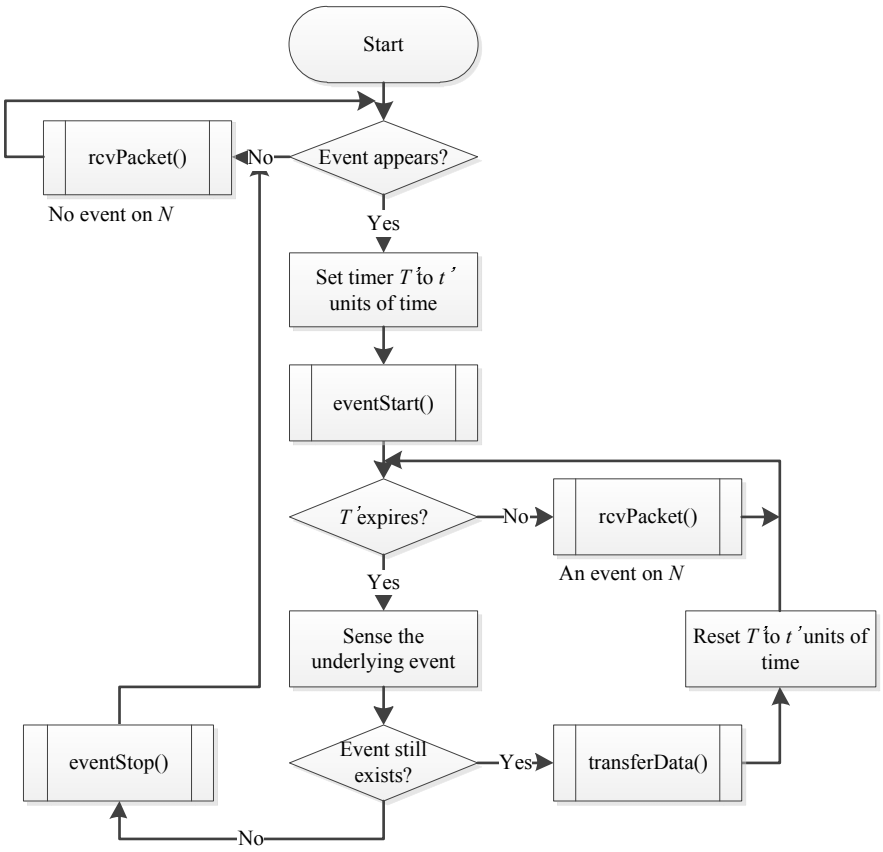


Figure A2. Flow chart for the operations performed by a node *N* other than the sink (*T'= packet size / data rate*; *1/T'* is the packet generation frequency)

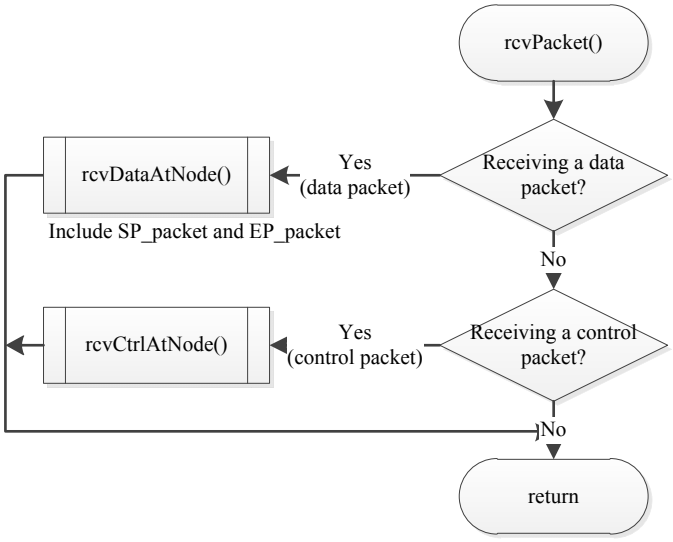


Figure A3. Flow chart for *N* on receiving a packet

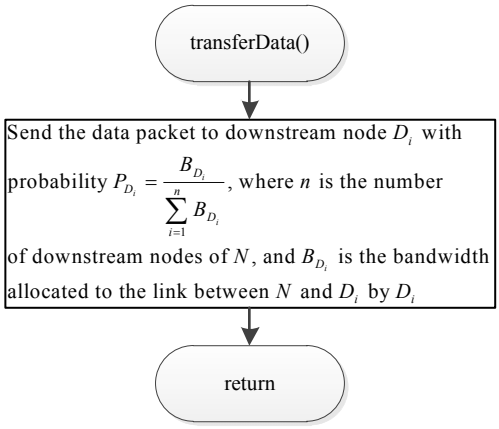


Figure A4. Flow chart for *N* to transfer a data packet

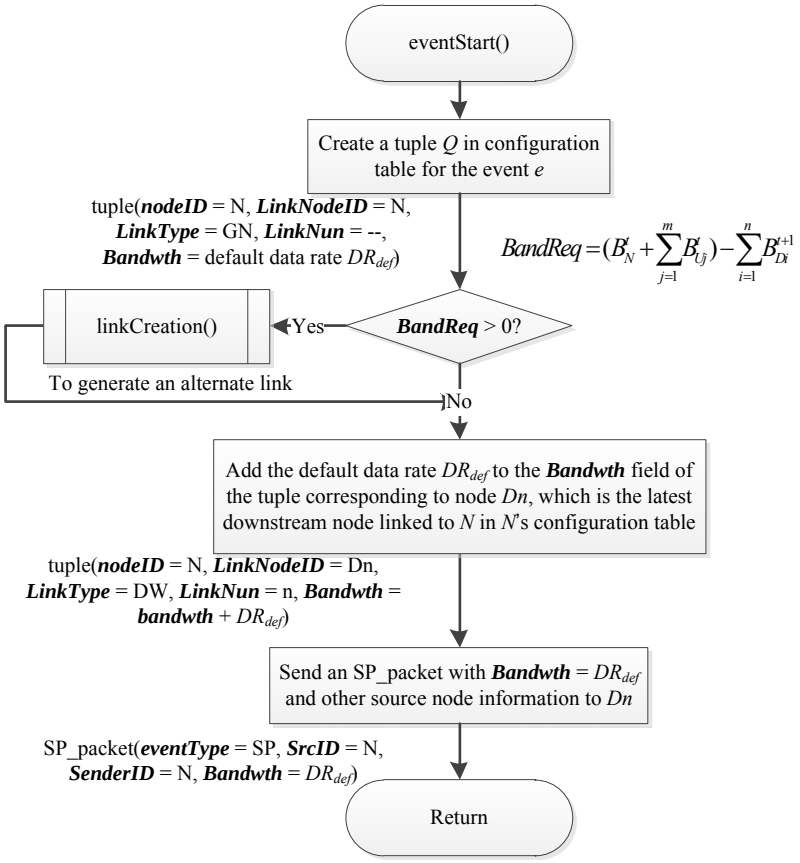


Figure A5. Flow chart for *N* on discovering an event

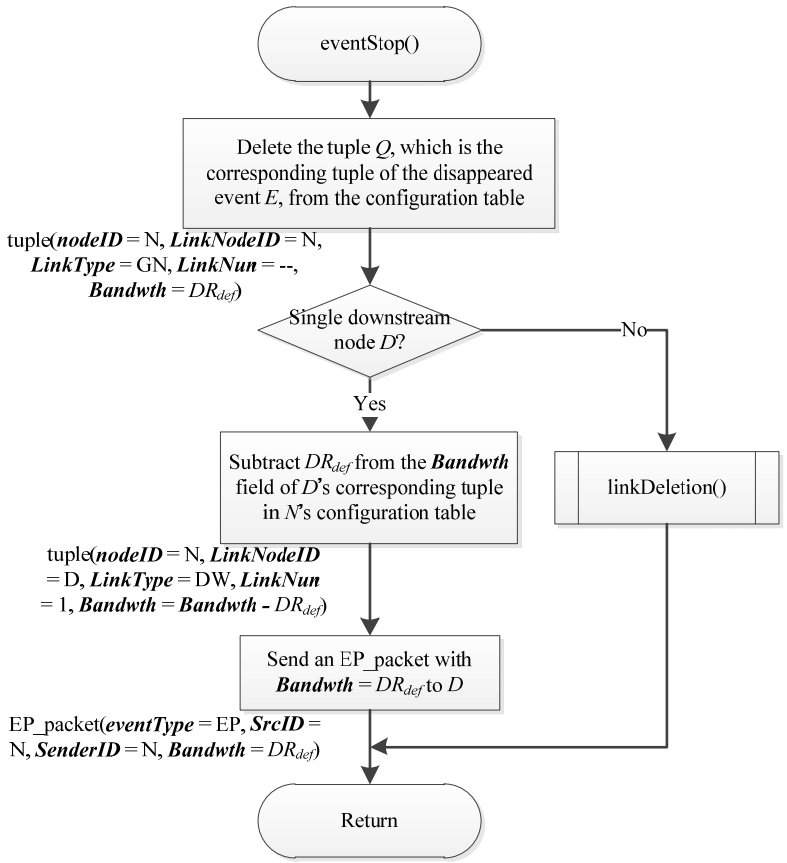


Figure A6. Flow chart for *N* on the termination of an event

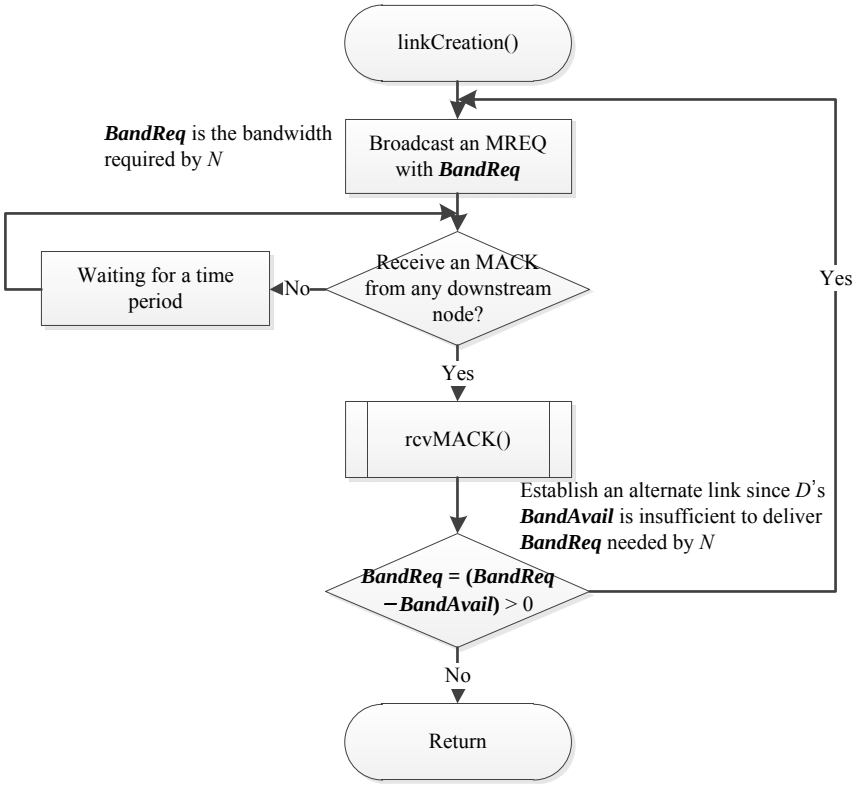


Figure A7. Flow chart for *N* to create multiple downlinks
(*D*: a downstream node; *U*: an upstream node)

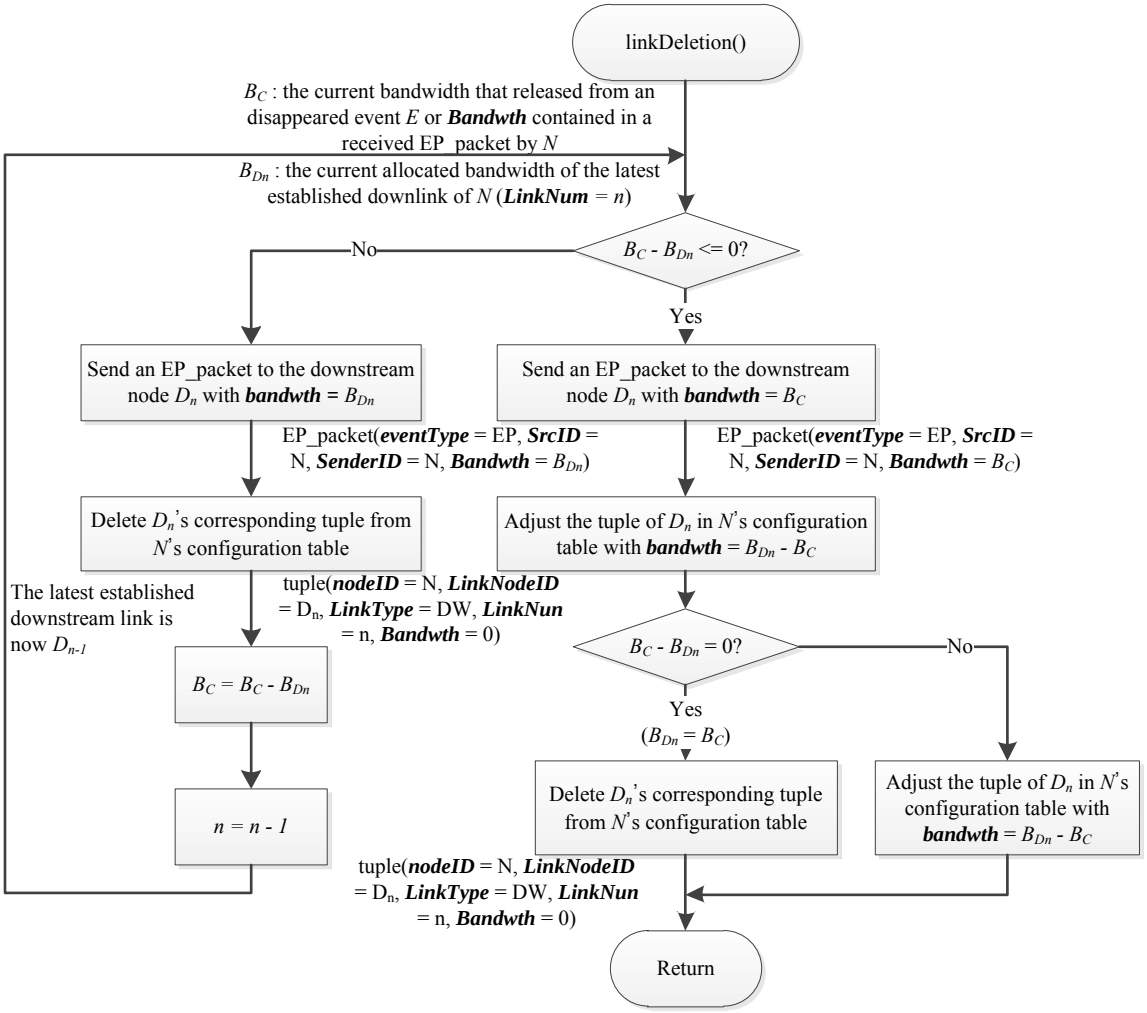


Figure A8. Flow chart for N to delete multiple downlinks

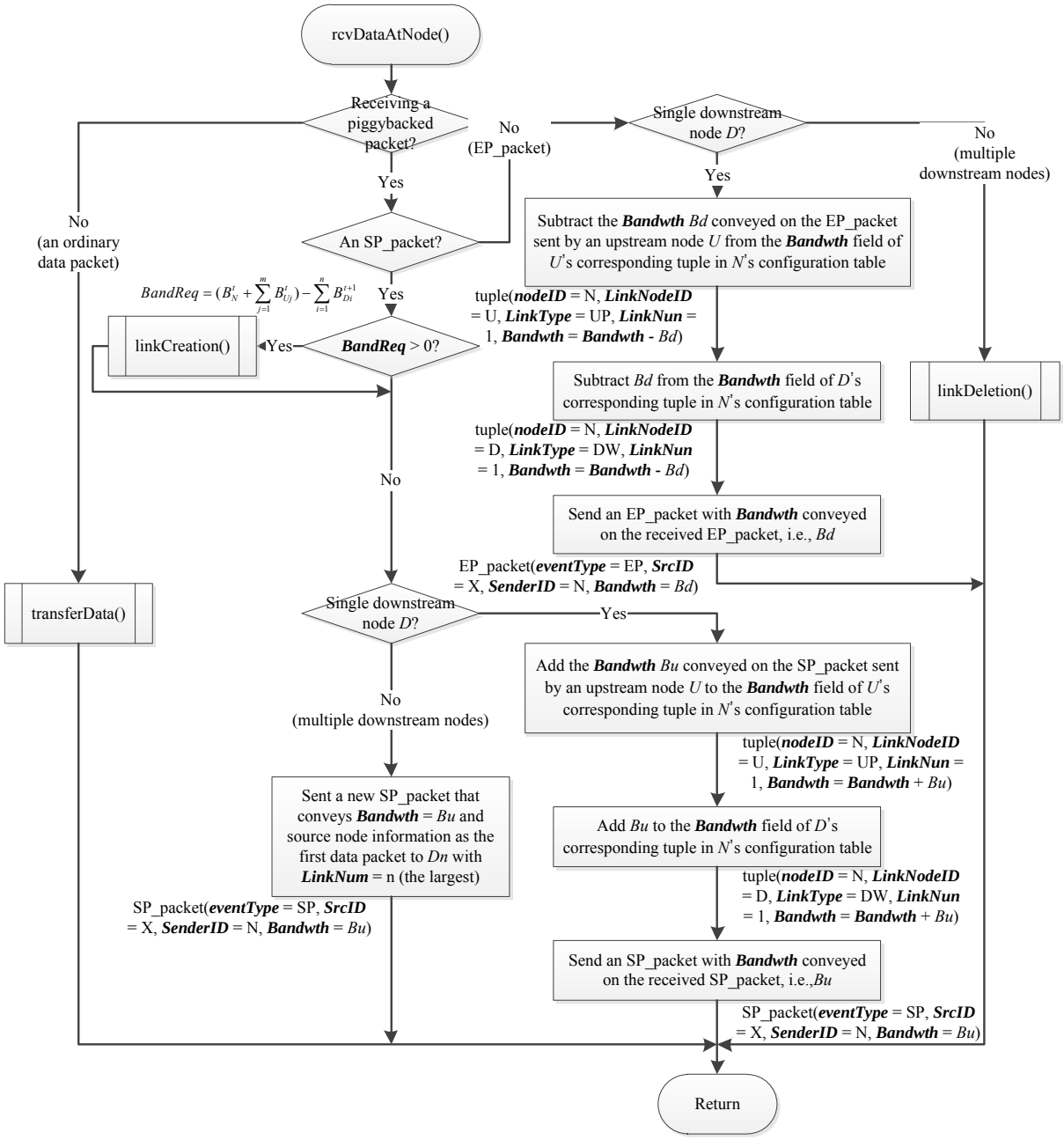


Figure A9. Flow chart for *N* on receiving a data packet

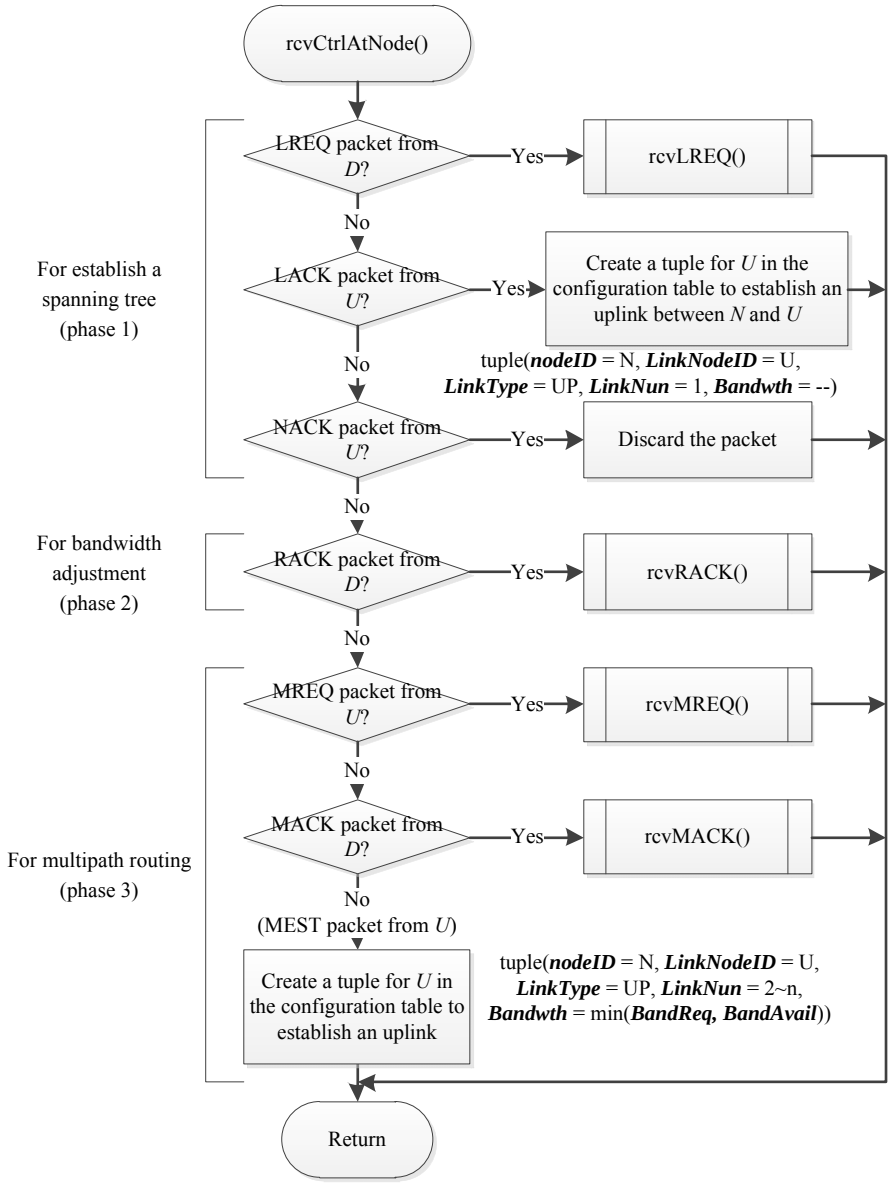


Figure A10. Flow chart for *N* on receiving a control packet
(*D*: a downstream node; *U*: an upstream node)

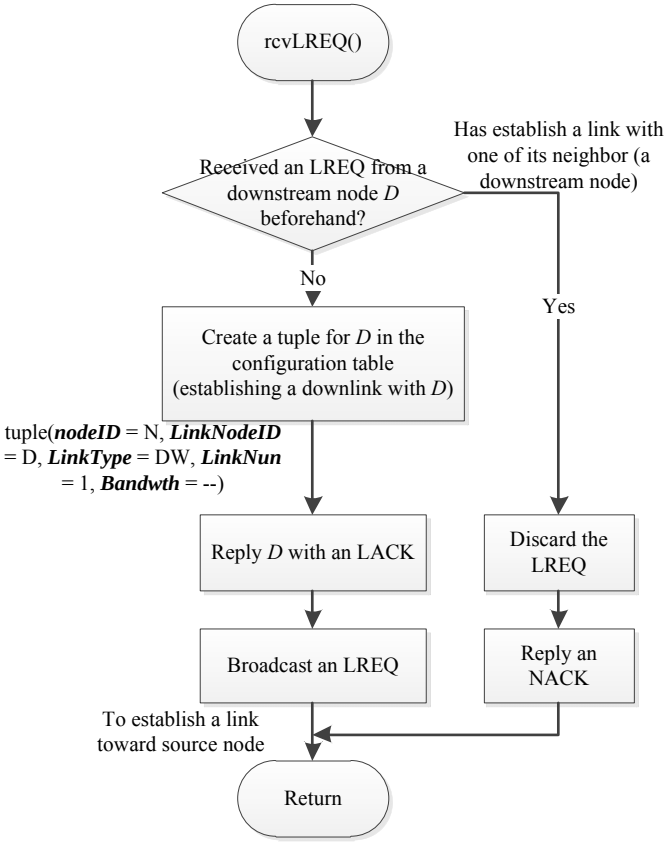


Figure A11. Flow chart for *N* on receiving an LREQ from a downstream node *D*

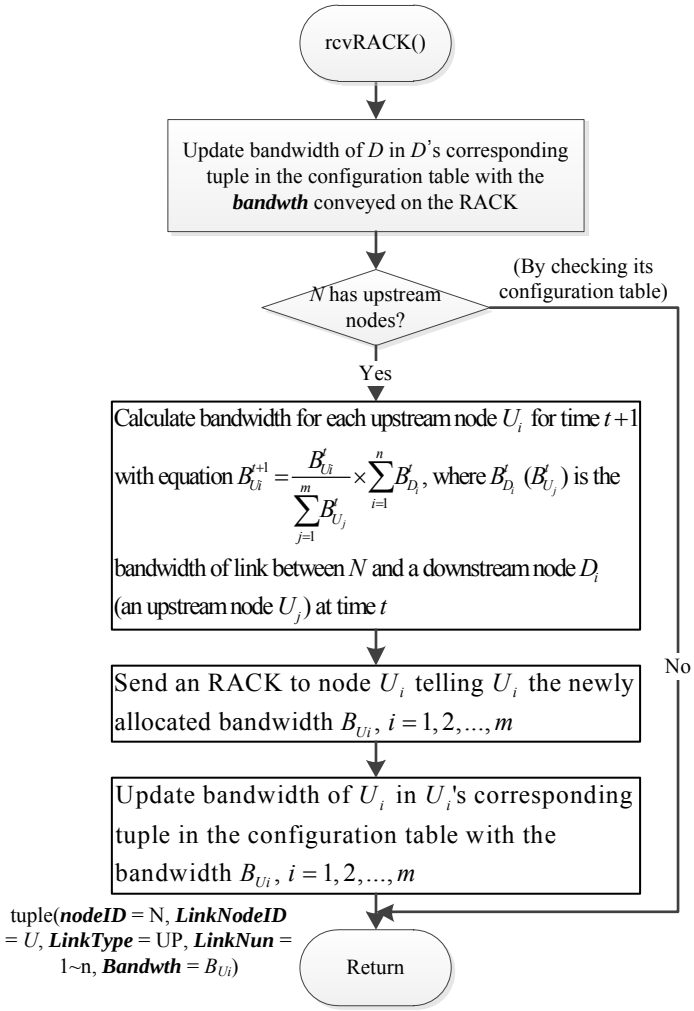


Figure A12. Flow chart for *N* on receiving an RACK from a downstream node *D*

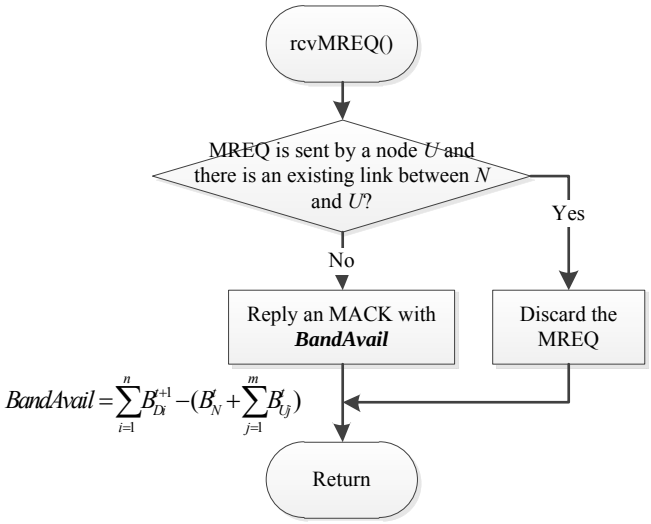


Figure A13. Flow chart for *N* on receiving an MREQ from a node *U*

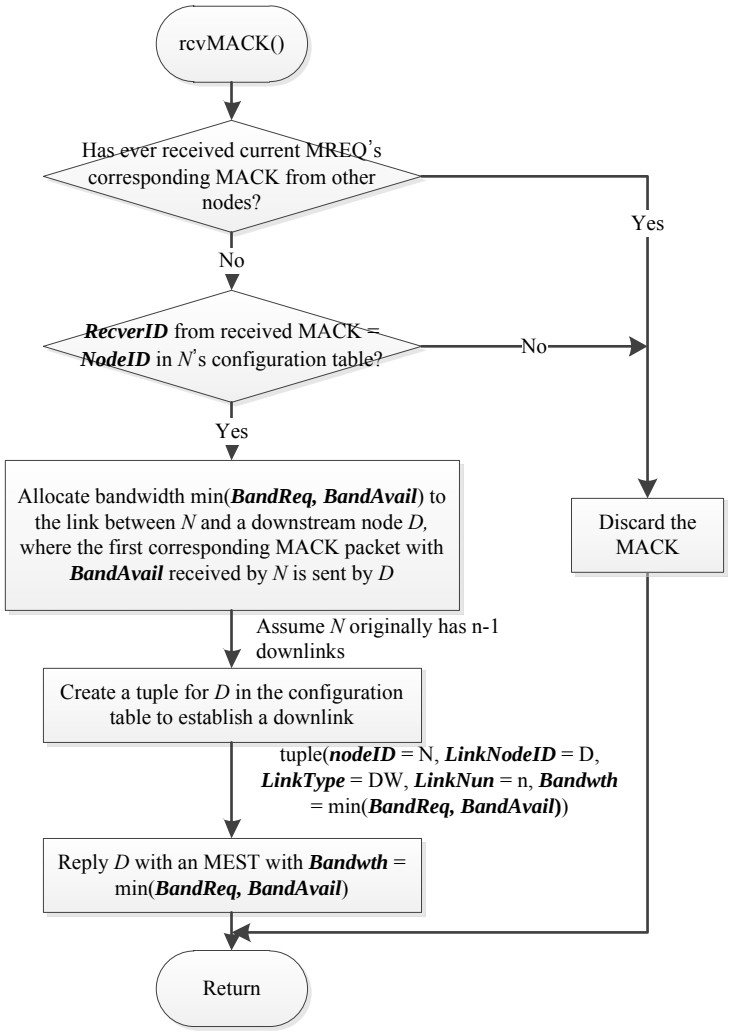


Figure A14. Flow chart for *N* on receiving an MACK from a node *D*