An Efficient Multicast Protocol Using de Bruijn Structure for Mobile Computing

David S. L. Wei and Kshirasagar Naik
Department of Computer Software
University of Aizu
Fukushima, 965-80 Japan
{d-wei, k-naik}@u-aizu.ac.jp

ABSTRACT

In this paper, we design a protocol to efficiently deliver multicast messages to mobile computers. The main concern in the design of such a protocol is to ensure that each message is delivered exactly once to each mobile host in a multicast group. However, the requirements of avoiding multiple delivery of a message, and of a host not missing a message are not easy to efficiently satisfy in a mobile environment. To satisfy these requirements, an earlier work had to actually broadcast a multicast message. The novelty of our approach is that we satisfy the multicast requirements without broadcasting a message, which is central to the efficiency of our protocol. We structure the mobile support stations (MSS) as a de Bruijn network, and define a multicast tree on the network. A multicast message is routed on a multicast tree on a de Bruijn network. Assuming that there are N MSS's in a network, a multicast group of mobile hosts belong to the cells of $\mathcal{N}$ MSS's, τ is the maximum of the time to route a message between two adjacent MSS's, Λ is the sum of the mobility rates of all mobiles in a multicast group, λ_{max} is the maximum among the mobility rates, and φ is the *cell cross-over time* of a mobile, our protocol runs with a message complexity of $O(min(\mathcal{N} \log N, N) + \Lambda \log N)$ and a communication delay of $O(\tau \log N + \lambda_{max}\varphi)$. Thus, our protocol is very efficient when a message is multicast to a small number of mobile hosts. To show the practicability of structuring a set of MSS's as a de Bruijn network, we have demonstrated how a de Bruijn network can be emulated as a collection of *virtual paths* on an arbitrarily connected ATM network.

Key words: Mobile network, multicast protocol, de Bruijn network, ATM network

1 Introduction

There is an enormous interest in network computing because of the availability of high-speed communication systems and cheap workstations. Wireless communication networks will lead to a proliferation of portable computers. Users will have the ability to retain their logical connections to the Internet even while being on the move. An architecture for mobile networking is shown in Fig. 1 [10]. A *mobile host* (MH) is one that can move while retaining its network connections. A static host with a wireless interface supporting a collection of MHs is called a *mobile support station* (MSS). The static hosts are interconnected by a fixed network. A geographical area served by an MSS is called a *cell*. Each MSS and the collection of MHs belonging to its cell form a *wireless network*. All the MHs belonging to a cell rely on the MSS of the cell for availability to the rest of the network. An MH can directly communicate only with the MSS of the cell to which the MH belongs. A fixed host without a wireless interface can also be viewed as an MSS whose "cell" is never visited by any MH.

In very simple terms, the problem of multicasting can be stated as follows: In a distributed system with N processes, a process sends a message to all the processes in a subset of the N processes. However, multicasting

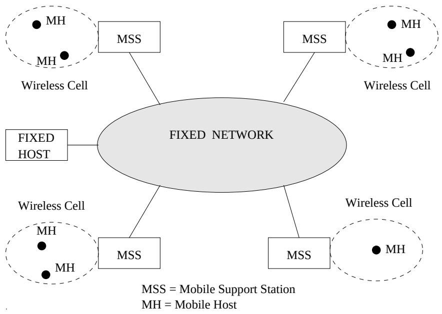

Figure 1: A mobile network

is a very broad term, and it can be found in all layers of protocols in a communication hierarchy and network-based computing. At the lowest level of communication, there are multicast packet-switching networks and multi-destination connections in ATM networks [7], [17]. At the network level, it is necessary to decide how connections should be routed within a network [3], [6]. Multicast service at the transport level are discussed in [13], [14]. At the application level, the idea of multicasting is used for process group communication [9], [11].

Different protocol layers address different aspects of data communication and distributed computing. Thus, the requirements of multicasting are highly context dependent. For example, at the network level some of the concerns are to meet constraints for parameters like delay, jitter, and link capacity [6]. At the transport level, the designers may focus on reducing the number of acknowledgement messages and retransmissions [14]. At the application level, multicasting becomes more sophisticated [9].

The idea of multicasting in a mobile environment is relatively new [1]. The two main issues in designing a protocol to deliver a multicast message to an MH are *multiple delivery* and *no delivery*. After receiving a message from one MSS, an MH may move to the cell of another MSS which has not yet transmitted the message to its MHs, and thus the newly arriving MH receives multiple copies of the same message. Before a message arrives at the current MSS, an MH moves to the cell of another MSS which has already transmitted the message to its MHs. Thus, the newly arriving MH does not receive the message. Also, if an MH moves to the cell of an MSS which is not known to the source MSS to service any MH in the multicast group, no multicast message will arrive at the new MSS leading to the *no delivery* problem. The problem of multiple delivery is easily solved by associating a sequence number with the messages. Multiple delivery of a message may occur even in traditional networks due to retransmission, and it is avoided by using a sequence number, say at the transport level. Though a similar strategy can be used in a mobile, single delivery has several advantages in a mobile environment, such as reducing the number of wireless messages, reducing interference among various wireless channels, and reducing power consumption in a mobile.

The problem of no delivery is a difficult one. It can be solved in a straightforward manner by sending a multicast message to *all* the MSS's in the network [1]. However, broadcasting a message to solve the inherent problems of multicasting in a mobile environment is not a sound idea. This is because the purpose of multidestination delivery, which is to prevent the source from making a number of unicast calls, let alone a broadcast, is defeated. The broadcast-based approach can not take advantage of situations where the multicast set is much smaller than the total number of MSS's in the system.

Acharya and Badrinath [2] have proposed a new protocol without using a broadcast. Their new approach essentially maintains a location directory for each multicast group. However, the number of messages needed for each multicast can not be easily counted for the protocol proposed in [2] due to the facts that there is a hidden cost of location management, the protocol ignores the communication pattern among MSS's, and, thus, also ignores the way to route a message to its destinations.

A desirable feature of a multicast protocol is to guarantee that a sequence of messages sent out by a source to a multicast group of destinations is received in the same order, as seen at the source, by all members of the group. This property is known as the *single source ordering* property [9]. In a mobile computing environment, the mobile hosts—and not the MSS's—are the real sources and destinations of multicast messages. Therefore, to guarantee the *single source ordering* property, we must consider the mobile hosts as the *sources* of the multicast messages in the multicast protocol. However, in the two multicast protocols presented in [1] and [2], the MSS's are assumed to be the sources of multicast messages. It can be easily shown that such an assumption can not guarantee the single source ordering property at the mobile host level. Consider two mobiles h_1 and h_2 sending multicast messages from the same cell, say the cell of MSS_i. Let h_3 belonging to the cell of MSS_k be a member of the multicast group receiving messages from h_1. Assume that h_1 asks MSS_i to multicast message $m1$, and moves to the cell of MSS_j, where it asks MSS_j to multicast message $m2$. If h_2 also asks MSS_i to multicast some messages before h_1 makes its request, and h_1 is the first mobile to make multicast request at MSS_j, then the protocols in [1], [2] will assign higher sequence numbers to $m1$ than to $m2$. Therefore, h_3 will receive $m2$ before $m1$. Such a scenario violates the single source ordering property seen at the mobile host level.

Now we summarize the contributions of this paper.

- Our protocol treats the mobile hosts as the sources and destinations of multicast messages, and guarantees the *single source ordering* property.
- By imposing a logical structure, namely the de Bruijn [15], on the nodes of the fixed network, our protocol guarantees the *exactly once delivery* property *without* using a broadcast technique.
- Another advantage of using a logical structure is that both message complexity and communication delay can be formally analyzed. We show that our protocol has a significantly lower message complexity and communication delay than the strategies in [1], [2].

In Section 2, we define a de Bruijn network, and justify its selection as a logical structure in our multicast protocol. An informal description of the protocol is given in Section 3. In Section 4, we formally present the protocol with its complexity. In Section 5, we show how a de Bruijn network can be realized as a collection of virtual paths on an ATM network. In Section 6, we summarize the merits and limitations of our protocol.

2 The de Bruijn Network

A directed de Bruijn network $B(d,n)$ has $N = d^n$ nodes with diameter[1] n and degree[2] d. A node v is labeled with a number with n digits $d_n d_{n-1} \ldots d_1$, where d_i is a d-ary digit. A node $v = d_n d_{n-1} \ldots d_1$ is *adjacent* to the node labeled $d_{n-1} \ldots d_1 l$, where l is an arbitrary d-ary digit. The $B(2,3)$ de Bruijn network is shown in Fig. 2. If node v is also adjacent to nodes $l d_n d_{n-1} \ldots d_2$, then the resulting network is undirected, denoted by $UB(d,n)$. The $UB(d,n)$ de Bruijn network is the same as $B(d,n)$ without selfloops and arrows.

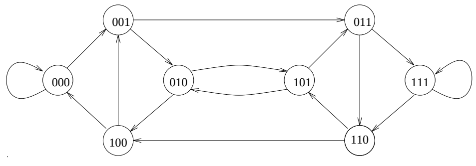

Figure 2: The B(2,3) network.

[1] The diameter of a network is the maximum of the distances between all pairs of nodes.

[2] The degree of a network is the maximum degree over all nodes, where the degree of a node is the number of links incident on the node.

Before presenting our multicast protocol based on de Bruijn networks, we discuss why the de Bruijn networks are attractive in efficient and fault-tolerant algorithm design. Consider an arbitrary network—not necessarily a de Bruijn network—with N nodes and m edges. Let Δ and D denote the degree and diameter, respectively, of the network. Then, the design of an algorithm depends on the topology of the network as follows [5], [8]:

- To minimize the running time of an algorithm, we must choose a D as small as possible.
- To minimize the number of messages for prefix computation related applications, etc., we need to minimize $(D+2)m$.
- If we are interested in fault-tolerance, we should have a large Δ.

For instance, one can obtain the best running time with $D = 1$, but in this case the network is complete and $m = N(N-1)/2$. The smallest value of m $(= N-1)$ is obtained when we organize the network as a linear array or a tree, but in the case of the linear array, D is also equal to $N-1$ and in the case of the tree, D is equal to $\log N$ but with a hot spot, say root of the tree.

In parallel and distributed systems, we should have the flexibility of having a large number of nodes, with few edges incident at each node, and a small diameter. This problem is known as the (Δ, D)-graph problem [8]. In the construction of such communication networks, there are three parameters of interest: N, Δ, and D. Specification of any two parameters will set a bound on the third one. For instance, for a given degree Δ and a diameter D, there is a theoretical upper bound, given by Δ^D, on the number of nodes in the graph. However, this upper bound is generally not achievable in many interconnection structures, such as hypercube and star graph. From the point of algorithm design, de Bruijn networks have two nice properties:

- In case of a de Bruijn network, given its diameter n and degree d we can achieve the upper bound on the number of nodes in the network. Thus, given N nodes we can construct a de Bruijn network by suitably striking a balance between its degree d and diameter n such that $N = d^n$.
- If we broadcast a message from one node to all other nodes, the message is sent between a pair of nodes *at most once*. A broadcast operation on a de Bruijn network with N nodes requires to pass at most $2N-1$ messages and suffers a communication delay (running time) of only $O(\log N)$. It may be argued that one can also design a broadcast protocol using, say a ring structure, to achieve a message complexity of $O(N)$. However, in such a case the delay complexity may shoot up to $O(N)$ in contrast to $O(\log N)$ for a de Bruijn.

Consider the network of Fig. 2. For illustration purpose, we represent a de Bruijn network as a multistatge network as shown in Fig. 3(a). It can be shown that given any pair of nodes $x = x_n x_{n-1} \cdots x_1$ and $y = y_n y_{n-1} \cdots y_1$, by shifting x n times with appropriate digit, say y_{n-i+1}, as new digit of rightmost bit at ith time, one can obtain a new address, say $y_n y_{n-1} \cdots y_1$, as desired. In other words, any node in the network is reachable from any other node in exactly n steps, although there might exist a shorter path. For example, referring to Fig. 3(a), in a multi-stage representation of a $B(2,3)$, there exists a unique path of length 3 between each node in the first stage (left most column) and each node in the fourth stage (right most column). In Fig. 3(b), we identify a (broadcast) tree from node 001 to all other nodes. For the sake of clarity, we do not show the other edges in Fig. 3(b). To perform a broadcast from node 001 we need to send the message at most once on each link of the tree.

Notice that the broadcast tree is just a logical view of the union of those paths of length n taken by the broadcast message. We use the broadcast tree as a means of explaining and analyzing our multicast protocol. Our protocol does not actually broadcast a message to all the nodes, rather it identifies a multicast tree on a broadcast tree depending on the multicast group of mobile hosts. Though our multicast protocol can employ a deBruijn network of arbitrary degree, for discussion convenience, we will employ the binary de Bruijn network (i.e. $d = 2$) in our protocol description.

Remark 1 Note that in order to avoid that a broadcast message traverses some link more than once, the protocol in [1] invokes a spanning tree algorithm to obtain a spanning tree rooted at the source node. Thus, if there are quite a few senders, then the spanning tree algorithm needs to be invoked many times, which results in more computation time and message complexity. On the other hand, our broadcast tree is implicitly constructed in a step-by-step manner from the definition of a de Bruijn network while routing a message from the source to the destinations.

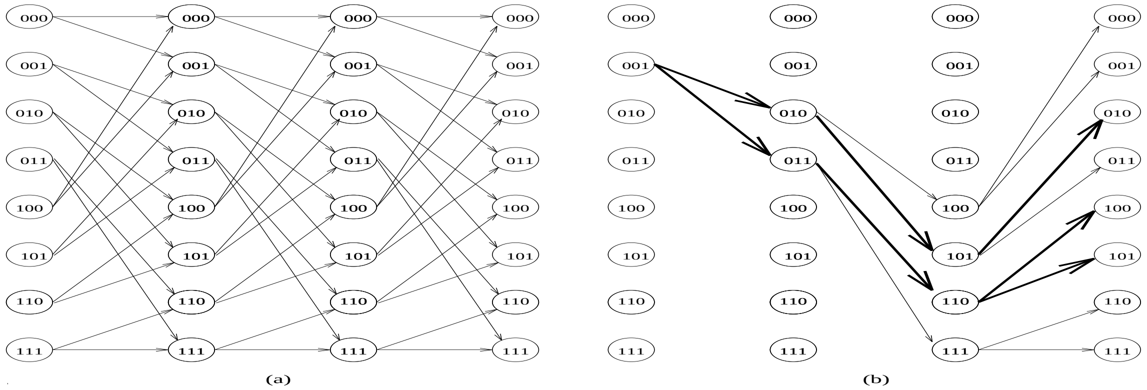

Figure 3: (a) Multi-stage representation of a B(2,3) network, and (b) The bold lines identify a multicast tree on a broadcast tree in the network.

3 Informal Description of The Multicast Protocol

In the multicast protocol, the mobile hosts are treated as the sources and destinations of multicast messages. The destination set of mobile hosts is called a *multicast group*. The set of MSS supporting those mobile hosts is also called a *multicast group*. The distinction between the two is made by using the terms mobile hosts and MSS. When we use the phrase "source MSS", we refer to the MSS supporting the source mobile of a message. We use m_{id} to denote the multicast message with identifier *id*.

The sender of a multicast message knows the identifiers of the mobile hosts in the multicast group. We assume that the source MSS also knows the multicast group of MSS's which either currently support a mobile host, or knows—directly or indirectly—an MSS supporting a mobile host in the group. The indirect knowledge is due to host mobility and the required hand-off procedure.

The protocol structures the mobile support stations as a de Bruijn network. We will explain the multicast operation on a network with eight MSS's. An arbitrary number of mobile hosts may be supported by this network. We can structure these eight MSS as a $B(2,3)$ de Bruijn network.

As an example, let node 001 be the sender of a multicast message, and let $\{010, 100, 101\}$ denote the set of destination multicast group of support stations. The corresponding multicast tree rooted at node 001, identified by the bold lines, is shown in Fig. 3 (b).

Because the addresses of the MSS's in a multicast group are known to the source MSS, the multicast tree is predetermined. Thus, the routing paths of the multicast message are known before the message is sent out from the source MSS. Each MSS on a multicast tree from the source to destinations keeps a copy of the message. The message is saved until the MSS is asked by the source to delete the message. The purpose of saving the message at each intermediate MSS is to efficiently send the message to a new MSS not belonging to the original multicast group, if an MH moves to its cell. If the new MSS of an MH (say h) is in the multicast set and has not yet transmitted the message, h will eventually receive the message, assuming that it stays long enough in the cell of its new MSS. Since host movement is a mechanical process, it may take several seconds or even minutes to enter and leave a cell. Therefore, this much time is enough for the hand-off procedure to be completed and any waiting message to be delivered to the mobile host.

In case the MSS of the new cell has already transmitted the message, the new MSS will know from the hand-off that h belongs to a multicast group, and retransmit the message once more for h. In case the new MSS does not belong to the multicast group when h moves in, the following are the two possible cases.

Case 1: The previous MSS receives the multicast message *before* performing a hand-off. During the hand-off procedure, the previous MSS also passes the *id* of the multicast message to the new MSS and notifies the new MSS that h is a destination MH of message m_{id}.

Case 2: The previous MSS receives the multicast message *after* it performs a hand-off. Sooner or later, the previous MSS will receive the message and transmit it to those destination MHs in its own cell. After checking the list of destination MHs, the previous MSS will notice that destination h has moved to a new MSS and then notify the new MSS of the message m_{id} needed by h.

For both the cases above, each MSS has to maintain some data structures such that the previous MSS knows the MSS of the new cell of h. Let mobile host h move from the cell of an MSS denoted by mss to the cell of an MSS denoted by mss'. While mss and mss' perform a hand-off for h, mss keeps some local information to remember that h has moved to mss'. Some mobile hosts might migrate so frequently that an MSS may have to accumulate too much of this local information. This local information will be deleted after a very long timeout T_L. In the following, we justify the deletion of local information after a long time out, and discuss the estimation of T_L.

Referring to Fig. 4, let a mobile host h be currently supported by station MSS_j. Since h can move to the cell of another support station, a station MSS_i is assumed to know the *approximate* location of h, rather than the exact station supporting h, in order to communicate with h. Thus, assume that at time t_j station MSS_i knows that h belongs to the cell of MSS_j. Let h move to the cell of MSS_l at time t_l. Now if we want MSS_i to communicate with h through MSS_l, then we must have a mechanism to inform MSS_i of the identifier of MSS_l at t_l after the hand-off between MSS_j and MSS_l. However, informing MSS_i whenever h moves to a new cell is not a good idea, because it will involve too many messages. An attractive alternative is to periodically inform MSS_i of the new location of h. In Fig. 4, h belongs to the cell of MSS_k at time t_k, and at t_k MSS_i is informed of the new location of h. Thus, during the interval $[t_j, t_k]$ station MSS_i communicates with h by sending messages to MSS_j irrespective of the actual station supporting h. When h moves from the cell of MSS_j to that of MSS_l, MSS_j knows the identifier of MSS_l after the hand-off. Thus, there exists an information chain from MSS_j to MSS_k during the interval $[t_j, t_k]$.

Now the question is how frequently MSS_i be informed of the new location of h. On one hand, if we inform MSS_i for every change in the cell of h, we may waste too many messages. On the other hand, if we inform MSS_i less frequently, the communication from MSS_i to h will go through a very long chain of support stations. Thus, a balance between a large number of messages to inform MSS_i and a long chain from MSS_i to h must be struck. Setting of the updation period, shown as T_L in Fig. 4, is a system parameter, which is outside the scope of this paper.

After a hand-off if the new MSS of a mobile host h finds that h needs to receive multicast messages, it sends a request message along the links of the broadcast tree in reverse direction toward the root of the tree. On the way to the root, the request message will reach an MSS, say mss'', containing the multicast message with the appropriate message *id*. Upon receiving such a request, mss'' will expand the multicast tree by sending the multicast message along the path taken by the request message. For example, in Fig. 5, if destination MH h_1 in the cell of MSS 101 moves to the cell of MSS 110 before MSS 101 transmits the message, MSS 110 will send a

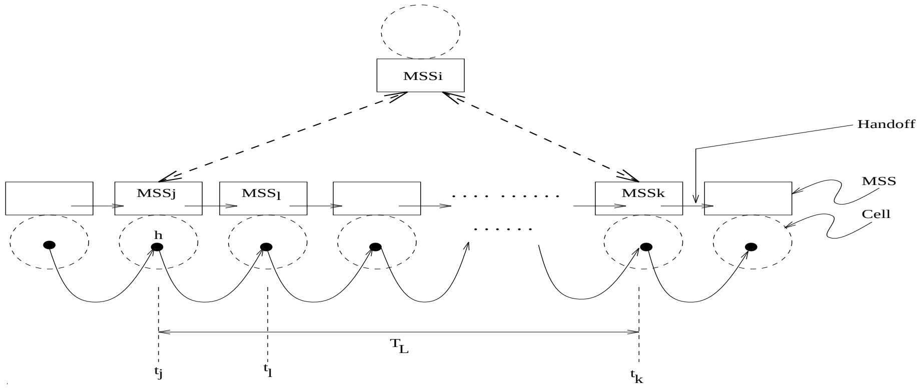

Figure 4: Periodic updation of the location of h at MSS_i.

request message along the dotted line toward the root of the broadcast tree. The request message will eventually reach the multicast tree at MSS 011. After receiving the request message, MSS 011 will send the multicast message to MSS 110 along the expanded multicast tree. Similarly, if destination MH h_2 in the cell of MSS 010 moves to the cell of MSS 011 before MSS 010 transmits the message, MSS 011 will send a request message along the dotted line toward the root and will cause MSS 101 to send the saved multicast message to MSS 011.

Regarding the issue of "multiple delivery", one might ask that according to the way we do multicasting, some MSS's, e.g. MSS 010 and MSS 101, can receive the multicast message more than once because they are repeated several times along a path of the multicast tree. A simple way to avoid this situation in our multicast scheme is to let each multicast message carry a counter with initial value 0. Each time a message is transmitted from one MSS to the next, the value of the counter in the message is increased by one. When an MSS receives a multicast message, it checks the value of the counter in the message. If the value is smaller than n, the height of the broadcast/multicast tree, the MSS simply passes the message to the next MSS; otherwise, the MSS is a destination and it transmits the message to the destination MHs in its cell. (It may be noted that the height of the broadcast/multicast tree is equal to the diameter of the de Bruijn network.)

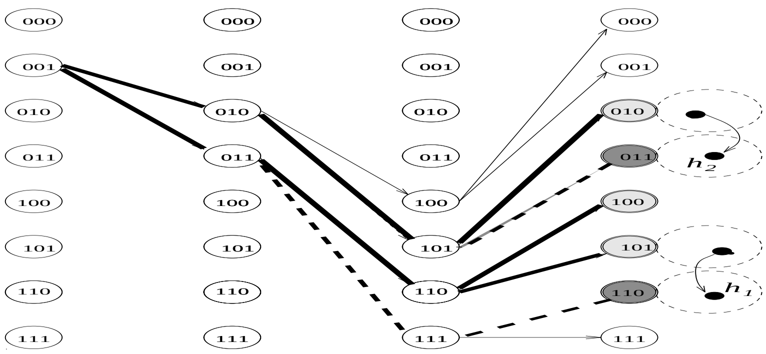

Figure 5: An expanded multicast tree with new edges (dotted ones) and new nodes (dark shaded ones).

4 Formal Description of the Protocol

4.1 Structure of the Protocol

The structure of the protocol is shown in Figure 6. The protocol consists of three subsystems, namely *wired*, *wireless*, and *hand-off*. The *hand-off* subsystem consists of two processes *initiator* and *responder*. These four processes communicate among themselves through message queues, namely $DregisterQ_{in}$, $DregisterQ_{out}$, $RegisterQ_{in}$, $RegisterQ_{out}$, $McastQ_{in}$, $McastQ_{out}$, $AckQ$, $GreetingQ$, $TransmitQ$, $MobileQ$, $RequestQ$, $NotifyQ_{out}$, and $NotifyQ_{in}$. The *wired* process on an MSS receives messages from the *wired* processes of the adjacent MSS's and the other three local processes. When the *wired* process receives a message, it decides whether to pass the message to the *wired* process of an adjacent MSS or the message is for local consumption. If a message is for local consumption, it is put in the appropriate queue.

The *wireless* process receives messages from the *wired* process through queues $McastQ_{in}$ and $NotifyQ_{in}$ and from the local mobiles through $MobileQ$. The message received from $McastQ_{in}$ is transmitted to some of the local mobiles. On the other hand, a message received from a local mobile is processed and put in $McastQ_{out}$, $AckQ$, or $GreetingQ$. When a new mobile enters the cell, it sends a greeting message which is put by the *wireless* process in $GreetingQ$. If a mobile acknowledges the receipt of a message, then depending on the local conditions, the *wireless* process inserts the identifier of the mobile into a list in $AckQ$. If a mobile wants to multicast a message, the *wireless* process puts the message in $McastQ_{out}$.

Now we explain the communication between the *initiator* on one MSS and the *responder* on another MSS. We have mentioned that when a new mobile enters a cell, eventually the *wireless* process puts a greeting message in $GreetingQ$. Assume that the mobile has moved in from the cell of MSS_j to the current support station MSS_i. Arrival of a message in $GreetingQ$ invokes the *initiator* process of MSS_i, which sends a deregister message to the *responder* process of MSS_j through $DregisterQ_{out}$. The *wired* process picks up the deregister message from $DregisterQ_{out}$, and forwards the message to the *wired* process of MSS_j. The *wired* process of MSS_j sends the message to its local *responder* through $DregisterQ_{in}$. Upon receiving a deregister message, the *responder* process of MSS_j generates a hand-off related message and puts it in $RegisterQ_{out}$. Eventually, the *wired* process of MSS_j sends the message from $RegisterQ_{out}$ to the *wired* process of MSS_i, which forwards the message to the *initiator* through $RegisterQ_{in}$.

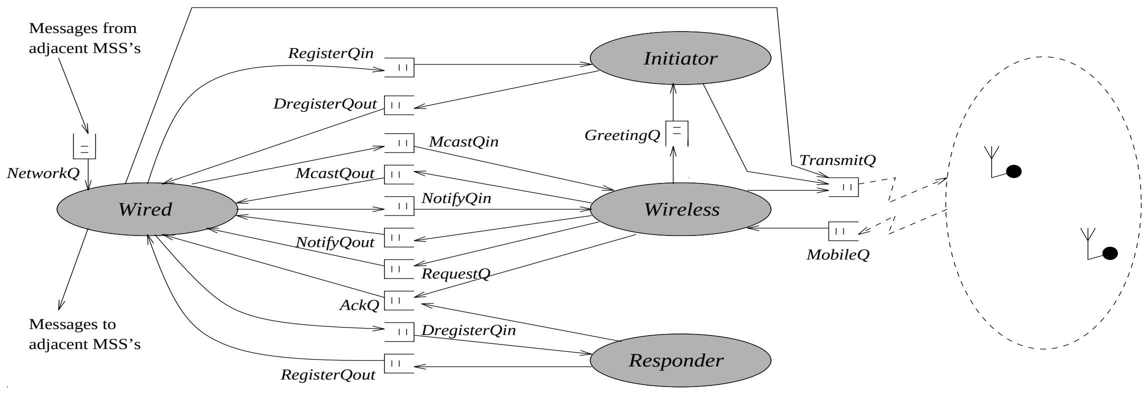

Figure 6: The structure of the multicast protocol.

4.2 Formal Presentation

We present our multicast protocol as a distributed one, i.e. each MSS individually and autonomously executes the presented protocol and communicates with other MSS's solely via message passing. The presentation of the protocol is with respect to MSS_i. Therefore, we explicitly use the index i in the protocol description.

4.2.1 Data Structures

We assume that the address of MSS_i is $x_n x_{n-1} \cdots x_1$; $n = \log N$, where N is the size of the fixed network. Also, let $m^q_{j,\hbar}$ denote the multicast message from mobile $\hbar$ with sequence number q originating from MSS_j. Thus, the tuple $(q, \hbar, j)$ forms a unique identifier of the message—j is essential to route the message. In the following, when we refer to a message in general, we use the variable m. On the other hand, a specific message is referred to as $m^q_{j,\hbar}$. The following ten types of messages are used in the protocol.

- $multicast(m^q_{j,\hbar})$: It contains the identifier $\hbar$ of source mobile, the identifier j of initiating MSS, sequence number q, $Dest(m^q_{j,\hbar})$ (defined below), the text of the message, and a counter, referred to as $multicast(m^q_{j,\hbar})(count)$ and with initial value 0, to be used in determining its routing path.

- $delete(m^q_{j,\hbar})$: A delete message for deleting message $m^q_{j,\hbar}$. This message contains all information similar to message $m^q_{j,\hbar}$ except the text.

- $request(m^q_{j,\hbar})$: A request message to ask for the multicast message $m^q_{j,\hbar}$. Its structure is similar to that of $delete(m^q_{j,\hbar})$ less $Dest(m^q_{j,\hbar})$.

- $ack(m^q_{j,\hbar}, h)$: The mobile h acknowledges the receipt of message $m^q_{j,\hbar}$.

- $Ack(m^q_{j,\hbar})$: When MSS_i sends this message to MSS_j, it means all the mobiles in $Local\_Dest(m^q_{j,\hbar})$ of MSS_i have acknowledged the receipt of message $m^q_{j,\hbar}$. This acknowledgement message contains the list of MHs which have acknowledged the receipt of the message $m^q_{j,\hbar}$. It also carries q, $\hbar$, j, and a counter with initial value 0.

- $ACK(q, \hbar)$: This message indicates that the multicast of the qth message from $\hbar$ is completed.

- $greeting(h, j, HQ, \hat{q})$: The greeting message is from mobile h which moved in from the cell of MSS_j. HQ is a list of pairs of the form $(\hbar, q)$ such that q is the sequence number of the latest message received by h from $\hbar$, which has not been acknowledged by h. $\hat{q}$ is the sequence number of the multicast message, sent out by h, which has not been acknowledged yet.

- $deregister(h, i, j, HQ, \hat{q})$: A message asking to deregister mobile h. This message contains h, i, and j, such that h has moved from MSS_i to MSS_j. HQ and $\hat{q}$ are identical to those in *greeting* message.

- $register(h, i, j, h\_RECD[], \{ACK(q, h)\}, MQ)$: A message asking to register mobile h. This message contains h, i, j, $h\_RECD[]$, $ACK(q, h)$, and MQ such that h has moved from MSS_i to MSS_j. The $\{ACK\}$ component, possibly empty, is a list of acknowledgements for h. Each member of $\{ACK\}$ is an ACK message for h. MQ is a list of 3-tuple of the form $(\hbar, q', j)$ such that $q' > h\_RECD[\hbar]$ is the sequence number of the message supposed to be received by h from $\hbar$ originating at MSS_j, which has not yet been received by h. It may be noted that the "$>$" relation is due to the fact that the expected *next* message from $\hbar$ to h is $h\_RECD[\hbar] + 1$.

- $notify(m^q_{j,\hbar}, h, k)$: MSS_i sends this message to MSS_k if message $m^q_{j,\hbar}$ arrives at MSS_i for mobile h, but h has moved to the cell of MSS_k. Also, MSS_k may in turn forward the *notify* message to MSS_l if h has moved from MSS_k to MSS_l, and so on.

Also, each MSS manages the following local data structures.

- $Dest(m)$: A list of pairs of the form (h, j), where h is the *id* of a mobile and MSS_j is the station whose cell contains h or knows where h is located, as explained in Section 3. If a message is destined for h at MSS_j, then the pair (h, j) is included on the list. This information is carried along with the multicast message.

- $Local$: The list of MHs in the cell of MSS_i.

- $Local\_Dest(m)$: A subset of $Local$ to receive message m.

- $ack\_list(m)$: A list of MHs which have acknowledged the receipt of message m. This is managed by the MSS that originated m on behalf of its source mobile.
- $local\_ack\_list(m)$: A list of MHs from the local cell, which have acknowledged the receipt of message m.
- $McastQ_{in}$: A priority queue of multicast messages. The *wired* module puts messages into the queue, and the *wireless* module selects messages for transmission based on a priority scheme. Given two messages $m^{q1}_{j1,\hbar 1}$ and $m^{q2}_{j2,\hbar 2}$, if $q1 = q2$ then the message with lower mobile identifier is given higher priority. Otherwise, the message with lower sequence number is given higher priority. Messages put in this queue are consumed by the *wireless* process. When messages are put in the queue, they are labelled "unmarked." The *wireless* process does not remove the messages from the queue—it simply labels the messages as "marked". The messages are actually deleted by the *wired* process from the queue upon receiving a delete request. The priority mechanism allows us to efficiently search the queue for some local consumptions. In the following, we use the same priority scheme for all the priority queues. However, the marking scheme does not apply to other queues except the $TransmitQ$.
- $McastQ_{out}$: A priority queue in which each entry holds a message to be multicast by MSS_i.
- $TransmitQ$: A priority queue of messages to be directly transmitted to the desired mobiles. The list $Local\_Dest(m)$ is attached to message m. ACK messages are also put in this queue. If an ACK message can not be delivered to a mobile, because it has moved out, the message is set to be "marked." The idea behind marking is to keep the message in the queue, until it is deleted by the *responder*, even if it could not be delivered to a mobile. However, in case an ACK message is delivered to the corresponding mobile, it is removed from the queue.
- $NetworkQ$: A priority queue containing all kinds of messages sent by the adjacent MSS's.
- $DregisterQ_{in}$, $DregisterQ_{out}$, $RegisterQ_{out}$, $RegisterQ_{in}$, $AckQ$, $GreetingQ$, $NotifyQ_{out}$, $NotifyQ_{in}$, $RequestQ$: Each of which is a priority queue.
- $mss\_temp(m^q_{j,\hbar})$: A variable to save message $m^q_{j,\hbar}$. Message $m^q_{j,\hbar}$ is saved if the counter in $m^q_{j,\hbar}$ is less than n, where n is the height of the multicast tree.
- h_{new}: The address of the MSS which either currently support mobile h or knows—directly or indirectly—an MSS supporting mobile h. An h_{new} is managed for mobile h which was once supported by MSS_i but moved out of MSS_i.
- $h\_RECD[\hbar]$: The *sequence number* of the last received multicast message from mobile $\hbar$ to mobile h.

4.2.2 Routing Scheme

In this subsection, we explain a routing scheme for communication among the *wired* processes. A *wired* process receives messages from both other *wired* processes and the local processes. Upon receiving a message, it must decide where to forward the message. The routing scheme basically computes the addresses of the required MSS's where the message needs to be forwarded. This computation is done using the information carried in a received message.

On the multistage representation of a de Bruijn network, some types of messages flow from a node on the first stage to the nodes on the last stage, while some other types of messages flow in the reverse direction. For the first kind of messages, an MSS may send a message out on a subset, possibly empty, of outgoing links, whereas, the second kind of messages are sent out on at most one link. Referring to Figure 3, assume that a message is to be multicast from a mobile in the cell of MSS 001 to some mobiles in the cells of MSS's given by the set

$\{010, 100, 101\}$. During the multicast process, an MSS on stage i finds out the address(es) of the desired MSS's on stage $i+1$ using the addresses of a subset of the destination MSS's. However, all types of messages are not routed in an identical fashion. We will explain the minor differences among their routing strategies. We classify the routing of the eight kinds of messages into three categories *Type 1*, *Type 2*, and *Type 3*, which are explained as follows.

Type 1: In this category, we route $multicast(m^q_{j,\hbar})$ and $delete(m^q_{j,\hbar})$ messages. These messages flow from the root to the leaves of a multicast tree. Here the routing is of *one-to-many* type, in the sense that an incoming message to an MSS may lead to messages on both the outgoing links. Given a set of destination MSS's, say Y, and value of a counter, say c, in an MSS X, we compute two addresses X' and X'' of the next MSS's to which the message needs to be forwarded. Either X' or X'', or both can be *null*. If both are *null*, then the message is locally consumed. Otherwise, the message is forwarded to the appropriate MSS(s). Before forwarding the message, the set Y is partitioned into two sets, Y' and Y'' such that Y' (Y'') is the subset of destination MSS's reachable from X' (X''). Computations of X', X'', Y', and Y'' are done using X, Y, and c as follows.

We assume that X is represented as $x_n x_{n-1} \cdots x_1$, and $\mathsf{Y} = \{\mathsf{Y}^1, \mathsf{Y}^2, \cdots, \mathsf{Y}^k\}$, where $\mathsf{Y}^i = y^i_n y^i_{n-1} \cdots y^i_1$.

- Computation of X' and X'': Let D be the set of $(n-c)$th bits of the elements of Y. (It may be noted that the value of c is in the range from 0 to n.) If $0 \in \mathsf{D}$, then $\mathsf{X}' = x_{n-1} \cdots x_1 0$. Otherwise, $\mathsf{X}' = null$. Similarly, if $1 \in \mathsf{D}$, then $\mathsf{X}'' = x_{n-1} \cdots x_1 1$. Otherwise, $\mathsf{X}'' = null$. It may be noted that $\mathsf{X}' = \mathsf{X}'' = null$ when $c = n$, which implies that the message is meant for local consumption by MSS X.

- Computation of Y' and Y'': $\mathsf{Y}' = \{\mathsf{Y}^i | \mathsf{Y}^i \in \mathsf{Y}$ and $(n-i)$th bit of Y^i is a $0\}$. Similarly, $\mathsf{Y}'' = \{\mathsf{Y}^i | \mathsf{Y}^i \in \mathsf{Y}$ and $(n-i)$th bit of Y^i is a $1\}$.

Type 2: In this category, we route $Ack(m^q_{j,\hbar})$, $ACK(q, \hbar)$, $deregister(h, i, j, HQ, \hat{q})$, $notify(m^q_{j,\hbar}, h, k)$, and $register(h, i, j, h\_RECD[], ACK)$ messages. These messages flow from some leaves to some other leaves or the root of the multicast tree. Specifically, $Ack(m^q_{j,\hbar})$ messages flow from leaves to the root, whereas the other messages flow among leaf nodes. However, these messages need not travel along the paths defined by the multicast tree, because these messages neither use nor modify the information in the internal nodes of the multicast tree. Therefore, it is possible to use less messages by not routing them along the multicast tree. Unlike the *Type 1* messages, a *Type 2* message flows from a source to exactly one destination MSS. Given the current MSS $\mathsf{X} = x_n x_{n-1} \cdots x_1$, a destination $\mathsf{Z} = z_n z_{n-1} \cdots z_1$, and a counter c, we compute the address of the next MSS X' as follows: $\mathsf{X}' = x_{n-1} \cdots x_1 z_{n-c}$.

Type 3: In this category, we route $request(m^q_{j,\hbar})$ messages. This type of messages flow from a leaf node toward the root along a path on a multicast tree. The flow of the message terminates at a node containing the requested message. The route of a *request* message is computed as follows. Given the current MSS $\mathsf{X} = x_n x_{n-1} \cdots x_1$, the root $\mathsf{R} = r_n r_{n-1} \cdots r_1$, and a counter c, we compute the address of the next MSS X' as follows: $\mathsf{X}' = r_{c+1} x_n \cdots x_2$.

4.2.3 The Protocol

In order to multicast a message, a mobile host makes a request to its MSS by sending the message. Successive messages from the same source mobile to the same multicast group follow the *window* flow-control mechanism [16]. Different mobile hosts belonging to the same cell can make independent multicast requests to the MSS. Thus, the multicast protocol running on the MSS can handle simultaneous multicast channels. In the following, we intuitively explain all the processes of the protocol and present their detailed behaviors in the appendix.

wired process: This process receives eight types of messages both locally generated and from the *wired* process on its adjacent MSS's. After receiving each message, it decides whether to forward the message to another MSS or to consume locally. Depending on local condition, it also generates an ACK message.

If it receives a multicast message from $McastQ_{out}$ or $NetworkQ$, it checks—using the counter—if the local MSS is a destination for the message. If so, it inserts the message into $McastQ_{in}$. Otherwise, it forwards the message to the appropriate MSS's. This is performed in Step 2.1.

If the process receives a *delete* message from $NetworkQ$, it checks if the message to be deleted is locally saved. If so, delete the saved message. Next, if necessary, it forwards the message to the appropriate MSS's. This is performed in Step 2.2.

If the process receives an *Ack* message from $NetworkQ$ or $AckQ$, it checks whether the current MSS is the destination of the message. If so, then update the list of mobile hosts acknowledging the corresponding multicast message. If all members of the destination multicast group have acknowledged, then it creates an ACK message to notify the source mobile, and creates a *delete* message to be propagated to all the MSS's in the multicast tree. Otherwise, if the current MSS is not the destination of the *Ack* message, then it is forwarded to the appropriate MSS. This is performed in Step 2.3.

If the process receives an ACK message from $NetworkQ$, it checks if the source mobile is in the local cell. If so, the message is put in the $TransmitQ$. Otherwise, the message is forwarded to the appropriate MSS. This is performed in Step 2.4.

If the process receives a $register$ message from $RegisterQ_{out}$ or $NetworkQ$, it checks if the current MSS is the destination of the message. If so, the message is put in the $RegisterQ_{in}$. Otherwise, the message is forwarded to the appropriate MSS. This is performed in Step 2.5.

If the process receives a $deregister$ message from $DregisterQ_{out}$ or $NetworkQ$, it checks if the current MSS is the destination of the message. If so, the message is put in the $DregisterQ_{in}$. Otherwise, the message is forwarded to the appropriate MSS. This is performed in Step 2.6.

If the process receives a $notify$ message from $NotifyQ_{out}$ or $NetworkQ$, it checks if the current MSS is the destination of the message. If so, the message is put in the $NotifyQ_{in}$. Otherwise, the message is forwarded to the appropriate MSS. This is performed in Step 2.7.

If the process receives a *request* message from *RequestQ* or $NetworkQ$, it checks if the requested message is locally saved in the MSS. If so, send the requested multicast message to the sender of the request message using *Type 1* routing scheme. It may be noted that due the *Type 1* routing, the requested message takes the same path as the *request* in the reverse direction. Otherwise, forward the request message toward the root of the multicast tree. This is performed in Step 2.8.

wireless process: This process receives five types of messages from queues $McastQ_{in}$, $NotifyQ_{in}$, and $MobileQ$. If it receives a *greeting* message from $MobileQ$, it simply puts the message in the $GreetingQ$. This is performed in Step 2.1.

If it receives an *ack* message from $MobileQ$, the sender of the *ack* is deleted from the *Local_Dest*. It then checks whether all members of *Local_Dest* have sent *ack* message. If so, then an *Ack* message is inserted in the *AckQ* queue. The idea behind generating an *Ack* message is to let the source of the corresponding message know that all the relevant mobiles of the current cell have received the multicast message. This is performed in Step 2.2.

If it receives a *multicast request* from a local mobile through $MobileQ$, then the multicast message must be propagated down the multicast tree defined by the destination MSS's. For this purpose, the *wireless* process prepares a $multicast(m^q_{i,\hbar})$ message, and puts it in the $McastQ_{out}$. This is performed in Step 2.3.

If it receives a *notify* message from $NotifyQ_{in}$, then the message is either forwarded to another MSS, along the chain of MSS's explained in Figure 4, or it is locally processed depending on the presence of the destination mobile in the current cell. A *notify* message arrives for a mobile if the mobile is to receive a multicast message. Therefore, if the corresponding multicast message already exists in the $TransmitQ$, then we simply include the identifier of the mobile in *Local_Dest*. If the message is not in $TransmitQ$, it checks whether the required multicast message is in $McastQ_{in}$. If the multicast message exists in $McastQ_{in}$, we have two possible cases.

In case 1, the message is "marked", and in case 2, the message is unmarked. The first case indicates that the multicast message has already been processed by the *wireless* process and transmitted to the local mobiles. Therefore, we need to retransmit the multicast message to the desired mobile. This is done by putting the multicast message in $TransmitQ$. In the second case, nothing needs to be done, because the multicast message will anyway be transmitted to the local mobiles including the desired one. Finally, if the multicast message exists neither in $TransmitQ$ nor in $McastQ_{in}$, then a request must be made so that desired multicast message arrives at the MSS. This request is performed by putting a *request* message in $RequestQ$. All these activities are presented in Step 2.4.

If it receives a *multicast* message from $McastQ_{in}$, it computes the set of identifiers of the mobiles to which the multicast message is to be transmitted. The corresponding $h\_RECD[]$ of each local mobile is also updated. The computed set of identifiers is put in *Local_Dest* and associated with the message text of the multicast message to be inserted in $TransmitQ$ for transmission. If a destination mobile has moved out to a new cell, then the new MSS of that mobile must be notified. This is done by putting a *notify* message in $NotifyQ_{out}$. This is performed in Step 2.5.

initiator process: This process receives two types of messages *greeting* and *register* from queues $GreetingQ$ and $RegisterQ_{in}$, respectively. If a *greeting* message is received, it puts a *deregister* message in $DregisterQ_{out}$. This is done in Step 2.1.

If it receives a *register* message from $RegisterQ_{in}$, it saves the $h\_RECD[]$, and adds h to *Local*. During the registration process, we handle two cases, namely h as a receiver and h as a sender. If h is a receiver of a multicast message, then we have *three* possibilities:

(i) The expected multicast message exists in $TransmitQ$. In this case, we simply update the $Local\_Dest(m^q_{j,\hbar})$ to include h. This is done in Step 2.2.a.

(ii) The expected multicast message has already been broadcast in the local cell. In this case, we need to rebroadcast the message for h. This is done in Step 2.2.b.

(iii) The expected multicast message never arrived at the MSS before. This happens if no other member of the *Dest* ever visited the cell of the MSS. Therefore, we need to send a *request* message to ask for the desired multicast message. This is done in Step 2.2.c.

It may be noted that the above three cases are mutually exclusive. On the other hand, if h has been sending multicast messages, there may be a need to send an ACK to h; this is done by checking whether the ACK component is non-null; if it is non-null, the ACK message is sent to h. This is done in Step 2.2.d.

responder process: This process receives only *deregister* messages from queue $DregisterQ_{in}$. Upon receiving such a message, h is removed from *Local* and h_{new} is set to the new MSS of h, and the following three steps are executed. First, we do the processing related to the pending acknowledgements from h. In fact, each pair of $(\hbar, q)$ in the HQ component of the *deregister* message represents an *ack* from h for the qth message from $\hbar$. Therefore, processing of the elements in HQ is identical to the Step 2.2 of the *wireless* process, and is done in Step 4. Second, in Step 5, we compute the set MQ of 3-tuples, where each tuple conveys information about a message for h arriving at the MSS but not yet delivered to h. This information, when carried in a *register* message, will be used in Step 2.2.c of the corresponding *initiator*. Finally, a *register* message is sent to the *initiator*. If an ACK message is pending in the $TransmitQ$ for h, then it is included in the ACK component of the *register* message, and is deleted from $TransmitQ$. This is done in Step 6.

The behavior of a mobile from the viewpoint of multicasting a message is explained as follows. As a sender, a mobile sends a multicast message to its local MSS subject to the satisfaction of the condition of the *window* flow-control mechanism [16]. As a receiver of a multicast message, after receiving a multicast message, a mobile sends an *ack* message to its local MSS. Thus, in the process of multicasting a message, a mobile does minimum work, and the actual protocol runs on the fixed network. This strategy leads to energy saving in a mobile, which is a desirable feature in mobile computing.

In data transfer protocols, the window flow-control mechanism is widely used. The size of a window is the maximum number of messages a sender can send before receiving an acknowledgement. From the viewpoint of the sender, sending a message reduces the window by one, and receiving an acknowledgement increases the window by one. As long as the window is more than zero, the sender can continue sending messages. As a special case, the alternating-bit protocol has a window size of one.

4.2.4 Proof of Correctness and Complexity Analysis

We first define a few terms. Let τ represent the maximum of the sum of the queueing time in a node and the transmission time on a link. The *mobility rate* of mobile h, denoted by λ_h, is the number of cell crossings by h per unit time. Λ is the sum of the mobility rates of all mobiles in a multicast group. λ_{max} is the maximum among the mobility rates in a multicast group. φ is the *cell cross-over time* of a mobile. Let there be N MSS's in a network, and let a multicast group of mobile hosts belong to the cells of $\mathcal{N}$ MSS's. For the proof of correctness and performance analysis of the protocol, we need the following assumptions and lemmas.

Assumption 1 *The time spent by a mobile in a new cell is enough for the hand-off to be completed and a message, if it already exists in the MSS of the cell, delivered to the mobile.*

Assumption 2 *The speed of a mobile through a chain of cells is lower than the speed of a message through the chain of MSS's of those cells.*

Assumption 3 *Throughout the execution of the multicast protocol, the multicast groups remain static.*

Now we explain the motivations behind these assumptions. If cells are very small and the mobiles move too fast, then the fixed network may not know even the approximate location of a mobile and an MSS may never be able to deliver a message to a mobile. Therefore, we need Assumption 1 to eventually deliver a message to a mobile. In case a mobile moves through a chain of cells not containing any other member of the multicast group, then we need to notify the MSS's of those cells that the mobile is supposed to receive a multicast message. Therefore, a notify message follows the mobile through the same chain of MSS's. Without Assumption 2, we will never be able to deliver a multicast message. The validity of these assumptions follows from the fact that electronic messages move much faster than the mechanical movement of the mobiles. Although there may be applications involving dynamic multicast groups, our Assumption 3 simplifys the presentation of the protocol.

Lemma 1 *The relative order of multicast messages, except the requested type, from the same source MSS is preserved when they arrive at the same destination MSS.*

Proof : According to our routing scheme, the path for each message from a source to a destination is fixed. □

Theorem 1 *Our multicast scheme guarantees that each destination MH receives a multicast message exactly once.*

Proof :
No delivery: For each multicast message, say $m^q_{j,\hbar}$, there are two kinds of destination MHs, namely nonmigrated MH and migrated MH, during the entire course of the multicasting of $m^q_{j,\hbar}$. It's obvious that each nonmigrated MH in $Dest(m^q_{j,\hbar})$ will eventually receive $m^q_{j,\hbar}$. For each of migrated MHs, say h, there are two possible cases:

Case 1: A mobile h in the multicast group travels through cells of MSS's not belonging to the multicast group. Initially, a member of the set of destination MSS's supports h or knows the approximate location of h. In this case, after the original MSS receives the multicast message, it will send a *notify* message along the chain of MSS's visited by h as stated in Step 2.5 of the *wireless* process. According to Assumption 2, the *notify* message will eventually catch up with h, and the corresponding MSS will send a *request* message asking for the desired multicast message. This is handled in Step 2.4 of the *wireless* process. If h stays in the current cell long enough, it will definitely receive the multicast message. However, if h keeps moving, in the worst case, all the MSS's not belonging to the original multicast set of MSS's will receive a copy of the desired multicast message, which will lead to h receiving the message.

Case 2: A mobile h in the multicast group eventually moves to the cell of an MSS in the multicast group. Now, we have two cases. First, the new MSS of h already contains the multicast message, in which case, according to Assumption 1, h will receive the message. Second, if the new MSS does not already contain the message and h moves out, then h will eventually receive the message similar to Case 1.

It may be noted that because of the updation of $h\_RECD[]$ after the delivery of a message, as specified in Steps 2.2 and 2.5 of the *wireless* process and Step 4 of the *responder* process, some messages may not be delivered to h. For instance, let $h\_RECD[\hbar] = q$ and let there be a multicast message $m_{j,\hbar}^{q'}$ at the MSS visited by h such that $q' > q+1$. According to the delivery condition, even if messages $m_{j,\hbar}^{q+1}$, $m_{j,\hbar}^{q+2} \cdots m_{j,\hbar}^{q'-1}$ are not present in the MSS, then message $m_{j,\hbar}^{q'}$ will be delivered to h, and $h\_RECD[\hbar]$ will be updated to q'. Such a situation means that the messages $m_{j,\hbar}^{q+1}$, $m_{j,\hbar}^{q+2} \cdots m_{j,\hbar}^{q'-1}$ will never be delivered to h. However, such *no delivery* problem would not arise due to Assumption 3 and Lemma 1. This is due to the fact that if message $m_{j,\hbar}^{q'}$ is present at an MSS, then all of the preceding undelivered messages must be present at the same MSS.

Multiple delivery: Each MSS in our multicast scheme maintains an array of integers $h\_RECD[1 \ldots \mathcal{H}]$ with every MH h, where $\mathcal{H}$ is the number of mobiles in the system. The value $h\_RECD[\hbar]$ denotes the *sequence number* of the *latest* multicast message received by h from $\hbar$. When h moves to a new cell, a message is possibly delivered to it only after the completion of the *hand-off* procedure. Therefore, while delivering a message the *wireless* process checks if the sequence number of the message to be delivered is greater than the sequence number recorded in $h\_RECD[]$. Since a multicast message with sequence number less than or equal to the sequence number recorded in $h\_RECD[]$ is never delivered, *multiple delivery* of a message does not take place. $\square$

Since we do not use broadcast mechanism to achieve multicast, host migration necessitates the *notify* and *request* messages. Also, the number of *notify* and *request* messages are proportional to the rate at which the migration takes place. These messages can be considered as penalty for host migration, and must be added to the message complexity of the protocol. Therefore, we define *mobility rate* of mobile h, denoted by λ_h, as the number of cell crossings by a mobile in unit time. Let

$$\Lambda = \sum_{h} \lambda_h,$$

where h belongs to a multicast group. Also, λ_{max} denotes the maximum among all λ_h of a multicast group. To compute the complexity of the communication delay, we define a *cell cross-over time*, denoted by φ, as the time taken by a mobile to cross a cell.

Theorem 2 *Our multicast scheme guarantees the single source ordering property.*

Proof: The proof follows from the facts that a mobile host is the source of multicast messages, and it assigns a sequence number to each message. $\square$

Theorem 3 *Let there be N MSS in the fixed network and let $\mathcal{N}$ be the number of MSS required to support the multicasting of a message. $\mathcal{N}$ includes both the original set of MSS and the new MSS due to the migration of mobile hosts. The multicast protocol requires $O(min(\mathcal{N} logN, N) + \Lambda \log N)$ messages to deliver a multicast message.*

Proof: By the time the protocol completes the multicast process, the multicast tree has $\mathcal{N}$ leaf nodes. Since the height of the tree is $logN$, there are at most $\mathcal{N}logN$ links in the multicast tree. A multicast message from the source to the destinations travel on a link of the multicast tree exactly once. A request message and a requested message travel on some links also once. A delete message from the source is similar to a multicast message. Each leaf node (destination node) on the multicast tree sends an acknowledgement message to the source. Therefore, there are as many messages as the number of links on the multicast tree. Thus, there will be at most five messages traveling on each link of the multicast tree. Due to host mobility, we need $O(\Lambda \log N)$ extra messages. Hence, proved. □

Theorem 4 *To multicast a message, the communication delay in our multicast protocol is* $O(\tau \log N + \lambda_{max}\varphi)$.

Proof: In case the members of the multicast group do not move, then the communication delay is given by $\tau \log N$. Host mobility simply increases the delay complexity by a factor of $\lambda_{max}\varphi$. The delay due to the *notify* and *request* messages and the *hand-off* procedure is absorbed in $\lambda_{max}\varphi$. □

5 A Strategy for Efficient Implementation

In this section, we discuss how our multicast protocol can be efficiently implemented. From the viewpoint of efficiency, we need to exploit the inherent concurrency among the *three* main activities, namely *routing*, *hand-off*, and *transmission* and *reception* of wireless messages. Since we have structured the protocol into four concurrent processes, as shown in Figure 6, we can obtain an efficient implementation. For instance, if multiple processors are available at an MSS, the four processes of the protocol can easily be scheduled on different processors.

An immediate advantage of running the *wired* process on a separate processor is that we obtain a smaller τ. This is because if all the above three activities are scheduled on the same processor or the protocol is designed as a monolithic structure, then a message will take longer to be routed through an MSS.

In order to extract the benefits of using a de Bruijn structure in the multicast protocol, we must devise a strategy to organize the fixed network as the desired structure. In reality, the fixed network could be a random one, rather than a regular structure. Therefore, directly running the multicast protocol on the fixed network may not be possible. However, assuming that the fixed network has been implemented using ATM switches, we explain how to realize a de Bruijn structure on the ATM network. As an example, in Figure 7, we show how to realize a de Bruijn network $UB(2,3)$ as a collection of *virtual paths* on an arbitrary ATM network [12]. In Figure 7(b), we have an ATM network with four switches and eight MSS's. We realize a link of the de Bruijn network as a virtual path on the ATM network. It is apparent that different virtual path layouts can be obtained for the same de Bruijn network.

Now we discuss the impact of realizing a network as a collection of virtual paths on the message complexity of and communication delay in the multicast protocol. Even though a virtual path may traverse multiple hops on the physical network, one message between two adjacent nodes on the de Bruijn network is represented by only one message on the corresponding virtual path. Therefore, the virtual path realization of a de Bruijn network has no impact on the message complexity of our multicast protocol. However, the value of τ is affected by a multiple of the maximum of hop counts among the virtual paths. Therefore, there is a need to reduce the maximum hop count in the virtual path layout. Another benefit due to reducing the hop counts is that imposing the logical structure on an arbitrary network would consume less network resources. Higher hop counts may lead to fewer sessions for other applications than if a multicast tree is constructed by other means. Obtaining the optimal virtual path layout for a given de Bruijn network on a given fixed network is beyond the scope of the paper.

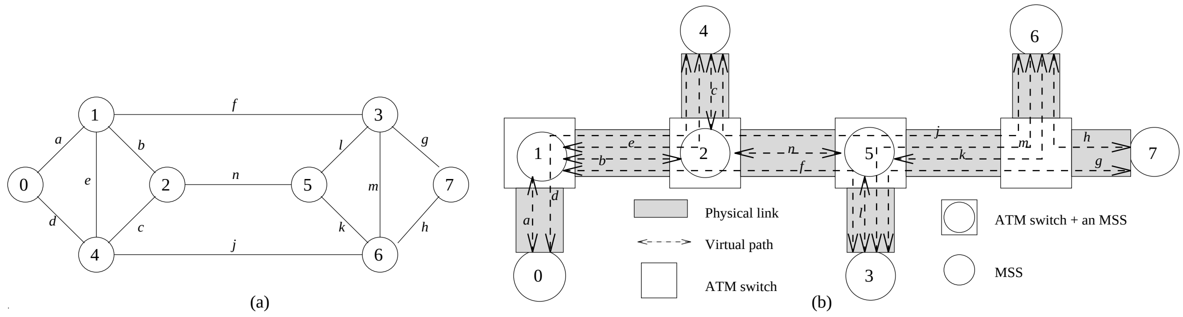

Figure 7: (a) UB(2,3), (b) A virtual path layout of UB(2,3) on an ATM network.

6 Concluding Remarks

Mobility of a host creates new problems in multicasting a message even for a simple requirement such as delivering the message to all the hosts in the multicast group. The problem of efficiently delivering a message to all mobile hosts in a multicast group is not a trivial one. The novelty of our approach is that to achieve a multicast, we neither use a broadcast technique nor depend on an external location management strategy [4]. Since we do not use an external location management strategy, there is no hidden cost in our protocol. Rather, the complexity of our algorithm is also a function of the mobility rates of the mobile hosts.

In this paper, we showed how by structuring the support stations as a de Bruijn network we can define a multicast tree of depth $logN$ for N MSS. The problem of no delivery is essentially solved by saving a message at the destination MSS and managing an information structure to keep track of the location of a mobile host belonging to a multicast group. This strategy makes broadcasting, to solve the same problem, unnecessary.

Our protocol runs with a message complexity of $O(min(\mathcal{N} \log N, N) + \Lambda \log N)$ and a communication delay of $O(\tau \log N + \lambda_{max}\varphi)$. The protocol in [1] has a message complexity of $O(N)$, while the protocol in [2] has a message complexity of $O(N)$ plus the hidden cost associated with external location management. In both of the later protocols, the communication delay is $O(\tau N)$ plus some hidden cost. Intuitively, for very high host-mobility, such that $\Lambda > \frac{N}{\log N}$ and $\lambda_{max}\varphi > \tau N$, our protocol incurs a message cost similar to, or even worse than, the cost of the protocol in [1]. The cost of our protocol is very inexpensive, when $\mathcal{N} \ll N$, which may be the case in general. The communication delay in our protocol is generally much better than the protocols in [1], [2] except when $\lambda_{max}\varphi > \tau N$. It may be noted that $\lambda_{max}\varphi \leq T_L$. Thus, by appropriately choosing T_L to be less than τN, we can ensure that our protocol has a better performance in terms of communication delay. The structuring of our protocol into four concurrent processes allows the *wired* process to be run on a separate processor. This can lead to a smaller τ than when the protocol is designed in a monolithic way.

From the viewpoint of complexity analysis, the advantage of using a de Bruijn structure is that we can accurately analyze the message and delay overheads without any hidden cost. Using an example, we have demonstrated how a de Bruijn structure can be realized using a set of virtual paths on an arbitrary ATM network. This shows that the idea of de Bruijn structure has not only a good theoretical foundation but also is easily realizable on popular networks such as ATM.

Designing multicast protocol for mobile networks is in a nascent state of research. Additional works may be done to address several problems such as *multiple source ordering*, efficient *virtual path layout*, and so on.

References

[1] A. Acharya and B. R. Badrinath. Delivering multicast messages in networks with mobile hosts. In *Proc. of the 13th. International Conference on Distributed Computing Systems*, pages 292–299, 1993.

[2] A. Acharya and B. R. Badrinath. A framework for delivering multicast messages in networks with mobile hosts. *Mobile Networks and Applications, Vol. 1, No. 2*, pages 199–219, 1996.

[3] L. Aguilar. Datagram routing for internet multicasting. *ACM Computer Communications Review, Vol. 14, No. 2*, pages 56–63, 1984.

[4] I.F. Akyildiz and J.S.M. Ho. Dynamic mobile user location update for wireless pcs networks. *Wireless Networks, Vol. 1*, pages 187–196, 1995.

[5] J.-C. Bermond and J. C. Konig. General and efficient decentralized consensus protocols. *Parallel and Distributed Algorithms, editors: M. Cosnard, et. al. North-Holland*, pages 199–210, 1989.

[6] M. Doar and I. Leslie. How bad is naive multicast routing. In *IEEE INFOCOM*, pages 82–89, 1993.

[7] K. Y. Eng and Y. S. Yeh. Multicast and broadcast services in a knockout packet switch. In *Proc. of the IEEE INFOCOM*, 1988.

[8] A. Esfahanian and S. L. Hakimi. Fault-tolerant routing in de bruijn communication networks. *IEEE Trans. on Computers, Vol. C-34, No. 9*, pages 777–788, Sept., 1985.

[9] H. Garcia-Molina and A. Spauster. Ordered and reliable multicast communication. *ACM Trans. on Computer Systems, Vol. 9, No. 3*, pages 242–271, August, 1991.

[10] D. Duchamp J. Ioannidis and G. Q. Maguire. Ip-based protocols for mobile internetworking. In *ACM SIGCOMM Symp. on Communication, Architectures and Protocols*, 1991.

[11] A. Scheper K. Birman and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. on Computer Systems, Vol. 9, No. 3*, pages 272–314, August, 1991.

[12] D. Medhi. Multi-hour, multi-traffic class network design for vp-based wide-area dynamically reconfigurable atm networks. In *INFOCOM'95*, pages 900–907, 1995.

[13] E. Pagani R. Aiello and G. P. Rossi. Design of a reliable multicast protocol. *IEEE INFOCOM*, 1993.

[14] K. K. Sabnani S. Paul and D. M. Kristol. Multicast transport protocols for high speed networks. In *International Conference on Network Protocols*, pages 4–14, 1994.

[15] M. R. Samatham and D. K. Pradhan. The de bruijn multiprocessor networks: A versatile parallel processing and sorting network for vlsi. *IEEE Trans. on Computers, Vol. 38, No. 4*, pages 567–581, April, 1989.

[16] N.V. Stenning. A data transfer protocol. *Computer Networks, vol. 1*, pages 99–110, 1976.

[17] S-C. Tu and W-H. F. Leung. Multicast connection-oriented packet switching networks. In *Proc. of the International Conference on Communications (ICC)*, 1990.

Appendix: Detailed Behavior of the Multicast Protocol

process *wired*
while ($NetworkQ$, $RegisterQ_{out}$, $McastQ_{out}$, $AckQ$, $NotifyQ_{out}$, $RequestQ$,
or $DregisterQ_{out}$ is not empty)
1 Pick up a message m from any of nonempty input queues;
2 **case** type of m:
2.1 *multicast message*: (*Denote m by $multicast(m^q_{j,\hbar})$.*)
if $(m^q_{j,\hbar}(count) < n)$
then
Keep $m^q_{j,\hbar}$ in $mss\_temp(m^q_{j,\hbar})$;
Increase $multicast(m^q_{j,\hbar})(count)$ by 1 and
forward the message using *Type 1* routing scheme;
else
Insert the message into local queue $McastQ_{in}$;
endif
2.2 *delete message*: (*Denote m by $delete(m^q_{j,\hbar})$.*)
if $(m^q_{j,\hbar}(count) < n)$
then
Delete $mss\_temp(m^q_{j,\hbar})$;
Increase $delete(m^q_{j,\hbar})(count)$ by 1 and
forward the message using *Type 1* routing scheme;
else
Delete message $m^q_{j,\hbar}$ from $McastQ_{in}$;
endif
2.3 *acknowledgement message*: (*Denote m by $Ack(m^q_{j,\hbar})$.*)
if $(i = j)$
then
Add the list of MHs carried by $Ack(m^q_{j,\hbar})$ to the list of $ack\_list(m^q_{j,\hbar})$.
if $(Dest(m^q_{j,\hbar}) = ack\_list(m^q_{j,\hbar}))$
then
Create a delete message $delete(m^q_{j,\hbar})$ and
send the message using *Type 1* routing scheme;
if $(\hbar_{new} = null)$
then
Insert $ACK(q,\hbar)$ into local queue $TransmitQ$;
else
Set $ACK(q,\hbar)(count)$ to 0 and
forward the message to MSS referred to by $\hbar$ using *Type 2* routing scheme;
endif
endif
else
Increase $Ack(m^q_{j,\hbar})(count)$ by 1 and
forward the message using *Type 2* routing scheme;
endif
2.4 *acknowledgement message*: (*Denote m by $ACK(q,\hbar)$.*)
if ($i =$ the destination address carried by $ACK(q,\hbar)$)
then
if $(\hbar_{new} = null)$
then
Insert $ACK(q,\hbar)$ into local queue $TransmitQ$;
else
Set $ACK(q,\hbar)(count)$ to 0 and
forward the message to MSS referred to by $\hbar$ using *Type 2* routing scheme;
endif
else
Increase $ACK(q,\hbar)(count)$ by 1 and

forward the message using *Type 2* routing scheme;
endif
2.5 *register message*:
(*Assume that the message is to be routed to MSS_j.*)
if ($i = j$)
then
Insert the message in local queue $RegisterQ_{in}$;
else
Increase counter by 1 and forward the message using *Type 2* routing scheme;
endif
2.6 *deregister message*:
(*Assume that the message is to be routed to MSS_j.*)
if ($i = j$)
then
Insert the message in local queue $DregisterQ_{in}$;
else
Increase counter by 1 and forward the message using *Type 2* routing scheme;
endif
2.7 *notify message*:
(*Assume that the message is to be routed to MSS_j.*)
if ($i = j$)
then
Insert the message in local queue $NotifyQ_{in}$;
else
Increase counter by 1 and forward the message using *Type 2* routing scheme;
endif
2.8 *request message*: (*Denote m by $request(m^q_{j,\hbar})$ and assume that the message was requested by MSS_r.*)
if ($m^q_{j,\hbar}$ is in $mss\_temp(m^q_{j,\hbar})$)
then
Send $multicast(m^q_{j,\hbar})$ toward MSS_r using *Type 1* routing scheme;
else
Increase counter by 1 and
forward $request(m^q_{j,\hbar})$ using *Type 3* routing scheme;
endif
end while
end process

process *wireless*
while ($McastQ_{in}$, $NotifyQ_{in}$, or $MobileQ$ is not empty)
1 Pick up a message m from any of nonempty input queues;
2 **case** type of m:
2.1 *greeting message*:
Insert the message in local queue $GreetingQ$;
2.2 *acknowledgement message ack*:
(*Assume that the acknowledgement is from h for message $m^q_{j,\hbar}$*)
Add h to $local\_ack\_list(m^q_{j,\hbar})$;
$h\_RECD[\hbar] \leftarrow q$;
Delete h from $Local\_Dest(m^q_{j,\hbar})$;
if ($local\_ack\_list(m_q^{j,\hbar}) \neq \phi$ and $Local\_Dest(m^q_{j,\hbar}) = \phi$)
then
Insert message $Ack(m^q_{j,\hbar})$, which contains the list of MHs
in $local\_ack\_list(m^q_{j,\hbar})$ into the local queue $AckQ$;
$local\_ack\_list(m^q_{j,\hbar}) \leftarrow \phi$
endif
2.3 *multicast request*:
(*Assume that the message is the qth message sent from mobile $\hbar$*)
Prepare $multicast(m^q_{i,\hbar})$ and insert it into local queue $McastQ_{out}$;
2.4 *notify message*:

(*Assume that the notify message is of the form $notify(m^q_{j,\hbar}, h, k)$*)
if ($h \in Local$)
then
  if (message $m^q_{j,\hbar}$ is in $TransmitQ$)
  then
    Update the message in $TransmitQ$ to include h in $Local\_Dest(m^q_{j,\hbar})$;
  elseif (message $m^q_{j,\hbar}$ is in $McastQ_{in}$)
  then
    if ($m^q_{j,\hbar}$ is "marked")
    then
      Set $Local\_Dest(m^q_{j,\hbar})$ to $\{h\}$ and
      insert $< (q, \hbar), message\ text\ of\ m^q_{j,\hbar}, Local\_Dest(m^q_{j,\hbar}) >$ into $TransmitQ$;
      (*The multicast message has already been broadcast in the local cell.
      The message needs to be transmitted once again for h.*)
    else (*The multicast message has not been processed by *wireless*.
      Eventually h will receive the message.*)
      Do nothing;
    endif
  else (*The message $m^q_{j,\hbar}$ is not in current MSS*)
    Insert $request(m^q_{j,\hbar})$ into local queue $RequestQ$;
  endif
else (*h is not in $Local$; In this case, h_{new} is not $null$*)
  Insert message $notify(m^q_{j,\hbar}, h, h_{new})$ into local queue $NotifyQ_{out}$;
endif

2.5 *multicast message*:
$Local\_Dest(m^q_{j,\hbar}) \leftarrow Dest(m^q_{j,\hbar}) \bigcap Local$;
$\forall h \in Local - Local\_Dest(m^q_{j,\hbar})$, $h\_RECD[\hbar] \leftarrow q$;
$\forall h \in Local\_Dest(m^q_{j,\hbar})$, **if** ($h\_RECD[\hbar] \geq q$)
then
  Delete h from $Local\_Dest(m^q_{j,\hbar})$;
if $Local\_Dest(m^q_{j,\hbar})$ is not empty
then
  Insert $< (q, \hbar), message\ text\ of\ m^q_{j,\hbar}, Local\_Dest(m^q_{j,\hbar}) >$ into $TransmitQ$;
endif
$\forall h \in Dest(m^q_k)$, **if** ($h_{new} \neq null$)
then
  Insert message $notify(m^q_{j,\hbar}, h, h_{new})$ into local queue $NotifyQ_{out}$;

endcase
end while
end process

process *initiator*
while ($GreetingQ$ or $RegisterQ_{in}$ is not empty)
1 Pick up a message m from any of nonempty input queues;
2 **case** type of m:
2.1 *greeting message* : (*Denote m by $greeting(h, j, HQ, \hat{q})$*)
Insert message $deregister(h, i, j, HQ, \hat{q})$ into $DregisterQ_{out}$;
2.2 *register message*: (*Denote m by $register(h, i, j, h\_RECD[], X, MQ)$*)
Add h into $Local$, and save $h\_RECD[]$;
2.2.a **for** each message $m^q_{j,\hbar}$ in $TransmitQ$, **satisfying** ($h \in Dest(m^q_{j,\hbar})$ and $h\_RECD[\hbar] = q - 1$)
Add h to $Local\_Dest(m^q_{j,\hbar})$;
endfor
2.2.b **for** each "marked" message $m^q_{j,\hbar}$ in $McastQ_{in}$, but not in $TransmitQ$,
satisfying ($h \in Dest(m^q_{j,\hbar})$ and $h\_RECD[\hbar] = q - 1$)
Set $Local\_Dest(m^q_{j,\hbar})$ to $\{h\}$ and
insert $< (q, \hbar), message\ text\ of\ m^q_{j,\hbar}, Local\_Dest(m^q_{j,\hbar}) >$ into $TransmitQ$;
(*The multicast message has already been broadcast in the local cell.
The message needs to be transmitted once again for h.*)
endfor
2.2.c **for** each entry $(\hbar, q', j)$ in MQ, **satisfying** ($q' > h\_RECD[\hbar]$)
Insert $request(m^{q'}_{j,\hbar})$ into local queue $RequestQ$;
endfor
2.2.d **if** ($X <> \phi$)
then
Insert each member of X into $TransmitQ$; (*X carries an ACK.*)
endif
endcase
endwhile
end process

process *responder*
while ($DregisterQ_{in}$ is not empty)
1 Pick up a message $deregister(h, i, j, HQ, \hat{q})$ from $DregisterQ_{in}$;
2 Delete h from $Local$;
3 $h_{new} \leftarrow j$;
4 **for** each pair of $(\hbar, q)$ in HQ
Add h to $local\_ack\_list(m^q_{j,\hbar})$;
$h\_RECD[\hbar] \leftarrow q$;
Delete h from $Local\_Dest(m^q_{j,\hbar})$;
if ($local\_ack\_list(m^{j,\hbar}_q) \neq \phi$ and $Local\_Dest(m^q_{j,\hbar}) = \phi$)
then
Insert message $Ack(m^q_{j,\hbar})$, which contains the list of MHs
in $local\_ack\_list(m^q_{j,\hbar})$, into $AckQ$;
$local\_ack\_list(m^q_{j,\hbar}) \leftarrow \phi$;
endif
endfor
5 Compute $MQ = \{(\hbar, q', j) | m^{q'}_{j,\hbar} \in McastQ_{in}$ and $q' > h\_RECD[\hbar]$ and $h \in Dest(m^{q'}_{j,\hbar})\}$;
6 **if** ($ACK(\hat{q}, h)$ is in $TransmitQ$)
then
Delete $ACK(\hat{q}, h)$ from $TransmitQ$;
Insert message $register(h, i, j, h\_RECD[], ACK(q, h), MQ)$ into $RegisterQ_{out}$;
else
Insert message $register(h, i, j, h\_RECD[], null, MQ)$ into $RegisterQ_{out}$;
endif
endwhile
end process