Université d'Ottawa · University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

MICIC, Aleksandar

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

Master of Computer Science

GRADE - DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

Initialization Protocols for TDMA in Single-hop Wireless Network

I. Stojmenovic

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

CO-DIRECTEUR DE LA THÈSE - THESIS CO-SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

M. Barbeau

D. Makaroff

J.-M. De Koninck, Ph.D.

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

SIGNATURE

DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Initialization protocols for TDMA in single-hop wireless network

Aleksandar Micic

Thesis submitted to the

Faculty of Graduate Studies

in partial fulfillment of

the requirements for the degree of

Master of Computer Science

Computer Science Department, SITE, University of Ottawa

Ottawa, Ontario, Canada

© Aleksandar Micic, Ottawa, Canada, 2003

National Library of Canada
Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services
Acquisisitons et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-90120-3
Our file Notre référence
ISBN: 0-612-90120-3

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canadä

Abstract

Although collision free TDMA schemes have been proposed and used for more than two decades, an important ingredient of these schemes, the initialization of stations (that is, assigning ID numbers 1,2,...,N) was not investigated until recently. In this thesis, we propose several new randomized and deterministic initialization methods, and measure the performance of these new and some known methods. The main contributions of this thesis are new randomized hybrid algorithms for the cases of known and unknown number of users. Performance of these algorithms was evaluated by comparing it with improved versions of existing algorithms, and an improvement from e·N to approximately 2.2·N was obtained. We also proposed the first deterministic initialization algorithms, and showed that they have comparable performance to the corresponding randomized algorithms. The initialization algorithms are then incorporated into collision-free TDMA schemes, which take into account the dynamic nature of network and dynamic bandwidth requirements.

Acknowledgment

I would like to express my deep gratitude to my supervisor Professor Ivan Stojmenovic who provided his guidance, encouragement and support during this research, reviewed my drafts of the thesis accurately to improve its contents and presentation.

1. Introduction 1

1.1. TDMA schemes and the initialization problem ... 1
1.2. Known algorithms ... 2
1.3. Contributions ... 2
1.4. Analysis ... 3

2. TDMA access schemes 5

2.1. The initialization problem ... 6
2.2. Collision-free TDMA schemes ... 7

3. Known initialization algorithms 10

3.1. *HNO-known-N* protocol ... 10
3.2. *HNO-unknown-N protocol* ... 11
3.3. Estimating the number of stations ... 12

4. New randomized initialization algorithms 13

4.1. A modification of *HNO-unknown-N* protocol ... 13
4.2. Simple initialization protocol with unknown, but estimated number of stations. 15
4.3. A hybrid protocol with known number of stations ... 17
4.4. A hybrid protocol with unknown, but estimated number of stations ... 21

5. Deterministic initialization protocols 28

5.1. A deterministic version of *HNO-unknown-N-mod* protocol ... 28
5.2. A tree-based deterministic protocol ... 30

6. Performance evaluation 42

6.1. Deterministic ... 42
6.2. Random-known ... 43
6.3. Impact of parameter *p* on *Hybrid-known-N* ... 43
6.4. Random-unknown ... 44

7. Dynamic assignments of channels 45

7.1. Station leaving with notice ... 45

7.2. Station leaving without notice ... 45
7.3. New station arriving ... 46
7.4. Channel borrowing ... 47
8. Conclusion ___ *52*
9. References ___ *54*
Appendix A - Source code ___ *56*
Appendix B – Detailed simulation results ___ *91*

1. Introduction

1.1. TDMA schemes and the initialization problem

This thesis studies TDMA access schemes that does not have random access schemes. There are various such schemes, like a very simple one with slotted access and static assignments of slots to the stations. We will give a short overview (Section 2.2.) of these schemes proposed in literature. We also propose one possible scenario for dynamic assignments of slots (Section 7). It discusses assignment of channels (broadcast slots), where stations can leave or join a group or they can borrow a channel.

Besides the access schemes overview, the thesis will primarily study one specific problem. To get a slotted scheme working the stations firstly have to initialize themselves, i.e. assign Id numbers: 1,2,3,... Once stations get their Ids they are assigned a transmitting slot for a static scheme. In dynamic slot assignments, a slot would not be assigned, but a station would still need an Id. For example, there are schemes where prior to a slot, a mini-slot is allocated used for handshaking (or similar) protocols. Than those mini-slots would be assigned to the stations based on their Ids.

There are two criteria to classify initialization protocols.

First, one can consider whether the protocol is deterministic or randomized. In first one, whenever a group of stations perform the initialization protocol, the assignment of Ids would be same. In section 5 we describe two deterministic initialization algorithms, which assume that each station has its unique serial number whose bits may be used for transmitting decisions. In randomized protocol, that assignment is completely arbitrary. In section 4 we propose several new randomized initialization algorithms.

Second, one can consider whether the stations know in advance what is the number of stations that participate in the protocol. Later, we will see how this affects the protocols. We will study both of them, but ultimately we always want a more generic case, i.e. when the number of stations is not known.

1.2. Known algorithms

Hayashi, Nakano, and Olariu [HNO] discussed randomized initialization protocol with both known and unknown number of stations. They did not consider deterministic version of protocols. For randomized ones they had another criteria of classification: type of the broadcast network. They considered cases when stations are able to recognize a collision on the network, and when station cannot distinguish silence from a collision. In our work we make assumption that the station are able to recognize a collision. We present (in Section 3.) two of their algorithms for known and unknown number of stations. Those two algorithms are completely different in their approaches and both of them have some advantages and disadvantage. The authors gave mathematical proofs of linear time complexities of their algorithms, and did not attempt to optimize the constants of linear proportionality.

1.3. Contributions

Related to initialization problem there are three aspects where we bring a contribution:

- enhance existing initialization protocols,
- introduce new hybrid initialization protocols,
- introduce a new type of deterministic protocols and propose solutions.

In [HNO], the protocol for unknown number of stations requires $8 \cdot N$ broadcast steps to initialize N stations. With a couple of modifications we will present a new protocol (Section 4.1.) that requires $e \cdot N$ steps. The known-number protocol was already optimized, so there is no need to performance change. However, we will introduce a flavor of that protocol for unknown number of stations (Section 4.2.), where the basic initialization process will remain same but the number of stations will be estimated prior to their initialization. There is no benefit of the protocol itself, but it will be used as a building block of another protocol.

Both protocols, even modified versions still have disadvantages. We found a way to merge the two and exploit only their advantages. The protocol will be presented as a new hybrid protocol. Actually we will first present known-number version of the protocol as a simpler one (Section 4.3.), and after that show how to modify that one for a more generic case, an unknown-number version (Section 4.4.) where the number of stations is estimated.

In section 5 we will describe two deterministic initialization protocols, which assume that each station has its unique serial number whose bits may be used for transmitting decision. One of the protocols is based on a modified version of randomized protocols. The basic algorithm remains same, and just the pseudo-random generator is replaced with one that depends on the unique serial number. It ensures the same sequence of serial numbers whenever the protocol is run and so the same initialization order for a given set of stations. We also introduced a completely new deterministic protocol that relies on fact that the serial numbers are unique and relatively short (32-bit long or so). Both of the protocols will be explained in Section 5.

In Section 7. we will describe one possible scheme of channel assignment in a dynamic environment. Having a group of stations initialize themselves, we will describe how a statation can join or leave the group. We will also describe how station with a variable and dynamic bandwith requirements can efficintly use a given slotted scheme. We use a well-known idea of mini-slots where stations anonunce their requirements. Stations have a queue where store that information and mutualy decide what will be a next station to transmit in the next slot. Our organization of the queue is novel that is supposed to be fair to all participating stations.

1.4. Analysis

While [HNO] proved linear time complexities of their protocols without measuring their performance and ignoring the constant in the linear function, we have implemented all the protocols and measured their performance (section 6.). Out of randomized protocols out new hybrid algorithms performed best. Approximate complexity is $2.2 \cdot N$.

The two deterministic algorithms have similar performance, with slight advantage of deterministic version of HNO protocol for a case of high number of stations. The complexity and absolute performance of deterministic protocols are coparable to those of non-deterministic version of protocols.

Although we focused on the performances, there are other perspectives of these protocols. They remain open for future work. Some of them could be robustification of the initialization protocols (recoverable from failures of stations) or implementing ones for networks with high delays (for example satellite networks).

For channel borrowing scheme, although we proposed a novel scheme, we did no implementation or performance analisys. This topic has already investigated in many reseach papers, so what we mostly did is just description of one possible scheme for the completeness of the thesis.

2. TDMA access schemes

A wireless network consists of N stations. A typical example is packet radio network, in which radio transceivers serve as stations. The network may or may not have a central shared resource, such as base station in cellular networks or a satellite. In a single-hop network, any station may directly communicate with any other station. Otherwise, the network is called multiple-hop one. In a single-channel network, all stations use the same channel for communication. In this thesis, we shall discuss only single-hop single-channel networks. Therefore, a message broadcast by one of stations on the common channel is simultaneously received by all other stations. If more than one of stations broadcast simultaneously over the channel, a collision occurs, which may be detected by all stations. Further, the silence (the lack of broadcast from any of stations) can be also detected by all stations. As customary, the time is assumed to be slotted and all the stations have a local clock that keeps (synchronous) time. Also, all broadcast operations are performed at time slot boundaries. The slots may be further divided into minislots. All stations run the same protocol.

Multiple access schemes and protocols are classified according to the bandwidth allocation mechanism, which can be static or dynamic, and the type of control mechanism being exercised. In general, multiple access schemes can be classified into the following three main categories: fixed assignment techniques, random access and demand assignment. In a fixed assignment technique, each slot or minislot is assigned to a particular station so that all stations are aware of the (mini) slot 'owner'. Such techniques perform well in heavy traffic circumstances. In a random assignment technique (such as ALOHA) [A, RS], stations broadcast when they have need, and at a time determined using a random function generator (thus stations can generate random bits). If collision occurs, the message is rebroadcast again at a later time. Such techniques are ideal for bursty traffic requirements. In the demand assignment schemes, stations are required to provide explicit or implicit information regarding their needs for bandwidth, which will be assigned on demand. Therefore, for variable bit rate (VBR) station bandwidth will be assigned according to burst length. Whenever the VBR station enters an idle period, the assigned bandwidth will be allocated to another station. Unlike fixed assignment, demand assignment schemes minimize wasted bandwidth by assigning bandwidth on demand. Also, unlike random assignment, demand assignment protocols avoid bandwidth wasted due to

collision by providing the connections with free contention bandwidth during active periods. MAC (medium access control) protocols based on demand assignment (sometimes called reservation access protocols) are the most appropriate for the requirements of integrated wireless networks [JM, KM]. In the packet reservation scheme [NGS], few slots are assigned to each host. If more slots are required, hosts compete for them by transmitting with some probability in each of unreserved available slots.

The purpose of access may be to broadcast a packet to all other stations (or few of them), or to get access to a central resource. In the later case, if the required information is of interest only to that station (e.g. patient information from a medical database), some of the schemes for time division as in the case of broadcasting packets may be applied. More precisely, fixed and demand assignments techniques may be appropriate while random access ones may not work in that context. However, if the required information is of interest to several stations (e.g. stock quotes) then it is more efficient that central resource periodically broadcasts that information to all stations (without waiting for any request). The broadcast frequency for each information is in correspondence to the expected demand for it, so that an average waiting time for the information is minimized. This problem is known as the scheduling broadcast problem and is extensively studied in literature [IVB]. In this thesis, we are not interested in this kind of multiple access problem.

Random access schemes for multiple access, like ALOHA, do not provide for collision free transmissions. In this thesis, we study fixed assignment techniques, i.e. the problem of designing MAC protocols that are collision free. The condition of collision free transmission is enforced on transmissions of actual data packets. We allow that, as a preprocessing or preparation step at the network start up time, or when adding or deleting stations, in order to organize transmissions, a collision may occur. However, collisions are not allowed to decide about transmitting each particular message, as in the demand assignment techniques.

2.1. The initialization problem

This thesis, in fact, deals with the mentioned preparation step. The main problem we study is scheduling transmissions so that each of N stations receives a unique identifier in the range from 1 to N, so that each station is assigned exactly one slot. Collisions are allowed in the process. In the network start up case, all stations are participating in the competition for ID

numbers at the beginning of the process. This problem of assigning to each of the N stations an integer ID number in the range $[1,N]$ such that no two stations are assigned the same ID is referred to as the *initialization* problem. The initialization problem is fundamental in both network design and in multiprocessor systems [OSZ]. If a station is to be added to the network, or deleted from the network, the ID numbers have to be modified. A straightforward solution is to repeat the initialization process whenever such a topological change in the network is detected by all stations (for instance, the signal for re-initialization may come as a result of a detected collision, which might be produced by an incoming station).

Once all stations are assigned ID numbers from 1 to N, each of them receives one of N time slots in which a collision free transmission of a packet is guaranteed. Such fixed assignment of time slots has satisfactory throughput if most of stations are busy transmitting the packets most of time. However, in many cases the traffic is bursty and such division into slots may be inappropriate for the integrated wireless networks because of inefficient radio channel spectrum utilization. It is widely accepted that the dominant services in the broadband environment are VBR services. Thus the length of each slot may vary, to improve the throughput. Another possibility is to keep the slot size fixed, which in some cases actually corresponds to the traffic characteristics (for instance, when messages are divided into equal size packets which are then transmitted individually). In such cases a station may borrow its own slot to another station if it does not have its own packet for that particular slot.

2.2. Collision-free TDMA schemes

Tobagi [T] gave a unified presentation of the various multiaccess techniques: fixed assignment, random access, centrally controlled demand assignment, and mixed strategies. This review is concentrated around distributed collision free methods, which are mainly TDMA (time division multiple access) based.

Collision-free multiple access schemes proposed in literature are based on the round robin method that is first proposed by Binder [B]. The hosts are assumed to have indices $1, 2, \ldots, N$ so that round robin is meaningful and efficient. However, none of the papers that deals with TDMA discussed the initialization problem, i.e. how nodes learn their host numbers (between 1 and N).

Kleinrock and Scholl [KS] described several multiple access schemes for radio networks. The network does not require any central station and hosts are assumed to be near each other, so that propagation delay is smaller than the packet length. In all their schemes, hosts have periodic access to broadcast. The next hosts to broadcast may be decided by using some alternating priorities, a random order, or a regular round robin conflict-free method. When a packet transmission ends, the alternating priorities rule [KS] assigns the channel to the same user who transmitted the last packet if he is still busy; otherwise the channel is assigned to the next user in sequence. This rule enables some hosts to 'privatize' the network if they have very long message queue. On the other hand, the rule exhibits the least overhead incurred in switching control between users. In the minislotted round robin scheme [KS], time is divided into minislots and slots. The length of a minislot is equal to the propagation delay. Minislots are assigned to hosts in round robin fashion all the time. That is, if host i used current minislot, the next minislot is used by host $(i \bmod N) + 1$, where N is the total number of hosts. If host i has a turn for the next minislot and has a packet, it starts to transmit its packet for the duration of the slot instead of minislot (thus minislot is replaced by a slot). Otherwise (if host i is idle), it does not transmit anything, and the silence is recognized by all hosts at the end of the minislot. This gives the signal to node $(i \bmod N) + 1$ for his turn. Authors [KS] show that their method is superior to TDMA, FDMA and random access schemes under heavy traffic conditions. At light input loads, random access scheme outperforms the minislotted round robin scheme.

Chlamtac, Franta and Levin [CFL] discussed BRAM method, which is almost identical to the minislotted round robin scheme. They introduce waiting time a, equal to propagation delay, for each node before it attempts to send a message. Each message contains node index, which enables other nodes to determine the waiting time, equal to a times the difference in the node indices. If the propagation delay is significant (which is the case in satellite networks), the methods may become inefficient.

Schoute [S] studied the problem of access control to a packet switched multi-access satellite broadcast channel in the framework of nonclassical control theory. It is shown that randomized decisions are not necessary, and for new packets only two modes of operation of the channel could be optimal. The optimal modes of operation are either All together (every station sends whenever it has a packet) or Round Robin (every station sends whenever it is its turn and it has a packet). Crossover arrival probability equations are also given.

Kosovych [K] examined two implementations of the fixed assignment technique for time sharing either a satellite or terrestrial channel among the users of a data communication system. In fixed assignments, transmission time on the channel is partitioned and permanently assigned to the users. In the first implementation, the users are cyclically ordered in the time sequence in which the users have access to the communication channel. Each user is periodically assigned a fixed time duration, during which the messages can be transmitted to the other users. In the second implementation, each access is of equal time duration but more than one access can be allocated to the terminals. The number of accesses assigned a user is the variable used to satisfy different requirements.

Rubin [R] discussed reservation and TDMA schemes for multi-access control. In a reservation scheme, channel bandwidth resources are appropriately divided into reservation and service components. The reservation component is used to process reservation request packets (following, for instance, slotted ALOHA protocol) while the service one is dedicated to message transmissions. In TDMA schemes, reservation time follows round robin access, in which hosts indicate whether they have packets to transmit. Packet transmissions then follow, also in round robin fashion, skipping nodes that did not have packets to send. In the round robin reservation scheme [B], packet slots are organized into equal size frames of duration greater than the propagation delay. One slot in each frame is permanently assigned to each station. To allow other stations to know the current state (used or unused) of its own slot, each station is required to transmit information regarding its own queue of packets piggybacked in the data packet header (transmitted in the previous frame). A zero count indicates that the slot in question is free. All stations maintain a table of all stations' queue lengths, allowing them to allocate among themselves free unassigned slots in the current frame. A station recovers its slot by deliberately causing a conflict in that slot which other users detect. For a station which was previously idle, initial acquisition of queue information is required and is achieved by having one of the stations transmit its table at various times. This scheme [B] was proposed for satellite channels.

3. Known initialization algorithms

The initialization problem has not been widely investigated in the past. Hayashi, Nakano, and Olariu [HNO] discussed cases with known and unknown number of stations and proposed algorithms for both cases that have linear complexity. They gave mathematical proofs of linear time complexities of their algorithms, and did not attempt to optimize the constants of linear proportionality. The paper [HNO] also described an initialization algorithm under different assumptions (stations cannot distinguish silence from collision) which is of no interest to this thesis.

The initialization problem was studied in [OSZ] in the context of incremental and dynamic construction of interconnection networks. Parallel algorithms on a number of network topologies were presented, including rings, meshes and hypercubes.

3.1. *HNO-known-N* protocol

Hayashi, Nakano and Olariu [HNO] proposed a simple and straightforward protocol (which will be referred to as *HNO-known-N* protocol) for the case when the number of stations, N, is known to all stations. Each station generates messages (at the beginning of the same slot) with probability $1/n$ (n refers to the current number of stations that are not initialized, while N is the number of all the stations. It is true that $N \geq n \geq 0$). This is the probability for which the chance of obtaining exactly one message on the channel is optimal. If exactly one message is transmitted, the unique station that transmitted the message (the winner) receives the ID gets assigned, n decrements by 1, and process continues recursively. Otherwise, stations retry the transmission. The process terminates when n reaches 1.

```
Protocol HNO-known-N
for n ← N downto 1 do {
  repeat
     station broadcasts on the channel with probability 1/n;
  until the status of the channel is single broadcast;
  the station that has broadcast is declared (N-n+1) station
};
```

The probability of having a unique message when n stations broadcast, each with probability $1/n$, is $(1-1/n)^{n-1} > 1/e$. Therefore the expected number of competition rounds before a 'winner' is found is $< e = 2.71\ldots$ The overall expected number of rounds is thus $< e \cdot N$. This is an important conclusion for our experiments. A more precise theoretical analysis is available in [HNO].

3.2. *HNO-unknown-N protocol*

The second protocol by Hayashi, Nakano and Olariu [HNO] deals with the case of unknown number of stations (protocol *HNO-unknown-N*). The protocol follows divide-and-conquer paradigm. Stations attempt to divide themselves into two non-empty groups, by transmitting messages with probability *1/2* by each station. One group continues recursively the 'competition' for ID numbers while the other one merely listens to detect the moment when the first group finishes the competition. Once a group size decreases to a single-station, the station gets assigned an ID, and the last pending group enters the competition.

```
Protocol HNO-unknown-N

Protocol Partition:
repeat
   each station selects 0 or 1 with probability 1/2;
   all the stations that selected 1 (first candidate partition) broadcast;
   let Status1 be the resulting status on the channel;
   all the stations that selected 0 (second candidate partition) broadcast;
   let Status0 be the resulting status on the channel;
until Status1 <> 0 and Status0 <> 0;
/* end Protocol partition */

/*begin protocol HNO-unknown-N */
l ← L ← nextId ← 1;      /* P(i): set of stations whose local variable l has value i */
while L >= 1 do {
   if |P(L)| = 1  then {
      the unique station in P(L) is declared the nextId-th station and quits the protocol;
      nextId ← nextId + 1; L ← L – 1 }
   else {
      use protocol 'Partition' to partition P(L) into two non-empty sets P(L) and P(L+1);
      L ← L + 1;
      each station in P(L) sets l ← L };
};
```

The status value · in the protocol 'partition' indicates silence, or empty selected group. The authors [HNO] showed that the main algorithm requires $2 \cdot N-1$ steps for checking $|P(L)=1|$ (N true and $N - 1$ false values in the **if** statement), and that with probability at least $1 - (1/2)^n$, the partition algorithm terminates in $8 \cdot N$ steps. The number of steps can be easily reduced, as we will show in the next section.

3.3. Estimating the number of stations

Willard [W] described two simple protocols for estimating the number of stations. In the first protocol, each station transmits with probability $1/2^k$ for $k = 1, 2, 3, \ldots$ until no collision on the channel is detected. The estimated number of channels is then 2^k, where k is the last tested value. The number of competition rounds to estimate the exact number of stations N is therefore $O(\log N)$. In the second protocol in [W], each station transmits with probability $1/2^L$, where $L=1, 2, 4, \ldots, 2^k$, until no collision on the channel is detected. Binary search between $L=2^{k-1}$ and $L=2^k$ is then performed to fine-tune the estimate. The number of competition rounds is $O(\log \log N)$. These protocols may be used in conjunction with any protocol for initializing stations with known number of them in case the number of stations is not known.

4. New randomized initialization algorithms

4.1. A modification of *HNO-unknown-N* protocol

The protocol *HNO-unknown-N* may be improved in two stages:

First, the protocol *Partition* may terminate an iteration as soon as it realizes that the first partition is empty.

Second, it is not necessary to add additional broadcast steps in the main initialization protocol in order to evaluate *P(L)*. The information can be inherited from Partition protocol, since all the stations in that group have already broadcast together (in the same time-slot). Stations just need to perform an additional computational step immediately after creation of a partition. Since the partition is non-empty, they just need to check whether the status of the broadcast was collision or a single broadcast. In the case of a single broadcast during the creation of the first sub-partition *P(L)* (represented by *partition1* in our modified algorithm), the unique station that broadcast immediately gets *Id* assigned. Similar *Id* assignment is done in case of a single broadcast during the creation of sub-partition *P(L+1)*. Once both sub-partitions broadcast, the next step should be decided. We distinguish the following cases.

- If both *partition1* and *partition0* had collision, then the partitioning process continues (parameter *L*, which represents the nested level of the current partitioning, is incremented). One of two partitions, say *partition1,* continues the process. Stations from *partition1* assign value *L* to its local level of partitioning, parameter *l*. During the next level of partitioning all the stations that have parameter *l* less that the current level *L* remain silent (because they are stations from pending *partition0*s).
- If any of partitions had a single broadcast but the other had collision, the single station gets assigned *Id,* while the collided stations continue partitioning. In this case there is no need to update parameters *L* and *l*, since no partition will go to pending state. Collided stations just continue partitioning at the same level.
- If both *partition1* and *partition0* had a single broadcast, there is no station left to continue partitioning. The current level of partitioning is decremented, meaning that the pending *partition0* with the highest level *l* reestablishes the process of partitioning.

After getting assigned *Id*, a station moves into a passive state, and still keeps listening to other competing stations. It is then able to update parameter *L*, which is needed to recognize the end of the protocol. The protocol ends when there are no more pending stations to reestablish partitioning. Since all the stations have parameter *l* initialized to *1* (and may increase or decrease it several times), the protocol ends when *L* reaches *0*.

Protocol HNO-unknown-N-mod

```
l ← L ← nextId ← 1;
station active;
repeat
   station that has l < L just listens and updates nextId and L;
   /* find partition 1 */
   repeat
       station selects 0 or 1 with probability 1/2;
       active station that selected 1 broadcasts;
       let Status1 be the resulting status on the channel;
   until Status1 <> 0;       /* exit if partition1 is non empty */
   if Status1=1 then {
        only station that sent in partition1: Id ← nextId and becomes passive;
        nextId ← nextId + 1 };
   /* find partition 0 */
   active station that selected 0  broadcasts;
   let Status0 be the resulting status on the channel;
   if Status0 = 1 then {
         only station that sent in partition0: Id ← nextId and becomes passive;
         nextId ← nextId + 1};
   if Status1 < 2 and Status0 < 2 then { L ← L - 1};
   /* if Status1 = 2 and Status0 < 2 then  L not changed /*
   /* if Status1 < 2 and Status1 = 2 then  L not changed /*
   if Status1 = 2 and Status0 = 2 then {
         L ← L + 1;
         station in partition1: l ← L };
until L=0;
```

Let us give a simple example for the protocol. Suppose that there are *5* stations to initialize. Broadcast steps are numbered, while pure computational steps are not:

- *5* stations to be partitioned; $l \leftarrow 1$; $L \leftarrow 1$;
1) *P1*: *3*; *3* stations randomly decide to broadcast in partition1
2) *P0*: *2*; *2* others broadcast in *partition0*
- *2* stations from *P0* pending with $l=1$
- $L \leftarrow 2$; *3* stations from *P1* get $l \leftarrow 2$ and are next be partitioned

3) *P1*: *2*
4) *P0*: *1*
- single station from *P0* get assigned $ID \leftarrow 1$
- no nested partitioning, no need to increase *L*, *2* stations from *P1* next to be partitioned
5) *P1*: *0*
- no need to evaluate *P0*, if *P1* is empty; retry partitioning
6) *P1*: *2*
7) *P0*: *0*
- *P0* is empty; retry partitioning; *L* unchanged
8) *P1*: *1*
- single station from *P1* get assigned $ID \leftarrow 2$
9) *P0*: *1*
- single station from *P0* get assigned $ID \leftarrow 3$
- no nested partitioning: *L* decremented, $L \leftarrow 1$; two stations from pending *P0* with $l=L$ (broadcast step 2) reestablish partitioning
10) *P1*: *1*
- single station from *P1* get assigned $ID \leftarrow 4$
11) *P0*: *1*
- single station from *P0* get assigned $ID \leftarrow 5$
- no nested partitioning; $L \leftarrow 0$; no pending stations with $l=0$ to reestablish partitioning; the end

4.2. Simple initialization protocol with unknown, but estimated number of stations

A group of unknown number of stations can be initialized by estimating the number of stations and using an initialization algorithm for the case of known number of stations. There are various techniques to estimate the number of stations. The overall performance of initialization algorithm for the case of unknown number of stations may be close to the performance in case of known number of stations, but, of course, not the same. We present one such protocol here, which as a basis for modification uses the simple initialization protocol with known number of users, i.e. *HNO-known-N*.

In addition to estimating the number of stations at the beginning, an efficient protocol needs to constantly update the estimation during the initialization process. If a collision occurs in several consecutive attempts, the number of stations is likely underestimated, and should be increased. Similarly, the estimated number of stations should be decreased in case of subsequent silence attempts. The update of the estimation can be passive (for example incremental: ± 1), but it can be also more aggressive (exponential, meaning that the first underestimate or overestimate

is updated by ± 1, but subsequent ones are updated by ± 2, ± 4, ± 8...). Although there are various ways of combining the estimation and the known-number-of-stations algorithm, the results do not vary much, as will be shown in the performance evaluation section.

We have selected the following version. The algorithm begins with a very passive initial estimation (number of stations is *2*), but has an exponential update of the estimation. The estimation process is incorporated into the initialization (main) protocol. The passive initial estimation does not affect much the performance, because of dynamic exponential update that depends on the outcome of each competition step. The estimation is not isolated from the initialization, because there are chances of assigning an *Id* during the estimation process. In the sequel, current estimated number of non-initialized stations will be denoted by m. As before, n will be the real current number non-initialized of station while N the number of all the stations. (The algorithm always tries to keep m to be as accurate as n).

It is not obvious to determine when a station exits the protocol, nor even to decide what it means to exit the protocol. If a station exits the protocol after assigning its *Id*, it has no information when all the remaining stations received their *Id*s, and will be unable to determine when to start the real usage of the communication channel and its slot just assigned! To solve the problem in our algorithm, at the beginning, we put each station in the active state, meaning that all of them fully participate in the assigning algorithm (that is, compete for *Id* by broadcasting a message with probability of $1/m$). Once the station gets assigned *Id*, it moves to a passive state where it simply keeps listening to other stations and accordingly updating local variables. By doing that, each station will know when the last station gets assigned *Id*. This will actually be the moment when all the stations exit the protocol.

Even less trivial to know is which station is the last one. If we had known the exact number of stations (N), the last station would be the one assigned exactly N as its *Id*. In the context of our protocol the last station is the one that broadcast a message while m (current estimated number of stations) is equal to *1*, and it happens to be the only station that broadcast. If $m=1$, stations broadcast with probability *1/1*, meaning that all the unsigned stations will broadcast. If it happens to be only one station broadcast, we are sure that our estimation of m was correct! That idea is actually implemented in the protocol, using variable "previous estimated m".

The initialization protocol can be specified as follows.

Protocol Estimated-N

```
m←2; station active;
step ← 0; lastId ← 0;
repeat
  prevM ← m;
  if station active then {
    station broadcasts on the channel with probability 1/m};
  switch status of the cannel { /* each station, no matter active or passive */
   case silence:          /* m is overestimated */
     if (step >= 0) then {         /* if previous step was underestimate ... */
       step ← -1}                  /*... just reinitialize the value of the update */
     else {
       step ←2* step;    /* for subsequent overestimates, double the update */
       m ← m + step;}
   case single:
     m ← m - 1;
     lastId ← lastId + 1;
     step ← 0;
     station that sent the message
        myId ← lastId;
        station passive;
   case noise:                     /* m is underestimated */
     if (step <= 0) then {
        step ← 1}
     else {
        step ←2* step;
        m ← m + step;}
  }; /* end of switch */
  if m < 1 and prevM <> 1 then { /* m should always be >0 */
      m ← 1 }
until m < 1 and prevM = 1; /* leave if previous estimation was 1 and now nobody is
left */
```

4.3. A hybrid protocol with known number of stations

By analyzing *HNO-unknown-N-mod* protocol in more details, one can observe that it performs as a sort of a "binary dividing protocol". It always tries to divide a current partition into two partitions. That process is recursively repeated until the number of stations in a partition reaches *0*. In the early phase of the protocol (the comment applies to both original and our modified version) the size of partitions is likely to be very large. When stations divide

themselves into two groups, there are fairly small chances that any of sub-partitions will be empty. For example, it is very unlikely that in the next step any of the sub-partitions of a partition of *100* stations will be empty. Nevertheless, algorithm performs in such a manner that for each step of partitioning at least two broadcast steps are necessary to ensure that the two new partitions are non-empty. Obviously, we are "loosing" almost unnecessary broadcast steps at early stage of the algorithm. Although we cannot avoid these kinds of verification broadcasts, we can force a faster partitioning. Instead of dividing themselves into two groups, stations can divide themselves into much larger number of smaller groups (for example, *2* or *3* stations per group). This way, the chances of retries will be still reasonable small; however, the chances of a unique broadcast will be significant. A station would not choose between 0 and 1 to decide which group they belong to, but, instead, any number from 0 to k at random (for example, k would be approximately $N/2$ or $N/3$).

Let us make the following two assumptions in the protocol:

- a perfect partitioning into same size sub-partitions,
- no retries.

In the first approach, it takes $2+4+\ldots+2^{N-L}$ broadcast steps to partition 2^N stations into 2^{N-L} groups of 2^L stations, while in the second approach it takes only 2^{N-L} such steps. Consider an example in Fig. 1 and Fig. 2 for N-$L=3$ (nodes of the graph are partitions and arrows are broadcast steps).

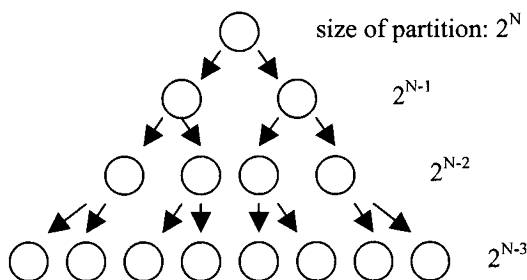

Fig. 1 Original approach (binary partitioning): 14

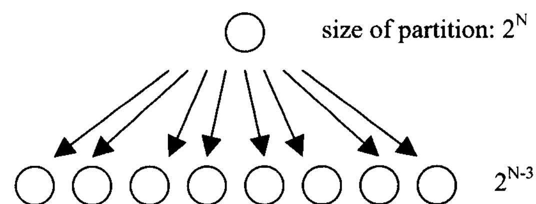

Fig. 2 New approach (k-ary partitioning): 8

Although we assumed an ideal partitioning, this new approach also offers some gain in non-ideal case. Same partitioning problems may occur in both former and new approaches, especially during the last step of the partitioning (in the example for the original approach, it is partitioning from size 2^{N-2} to 2^{N-3}). That means, for example, that retries will be likely while

creating a partition of size 2^{N-L} (especially if that is a small number), for both approaches. There are no other major differences between binary partitioning into groups in the original approach and directly partitioning a group of 2^N stations into 2^{N-L} sized groups. Still the performance evaluation in later section will show some advantage for the protocol that divides into smaller sized groups. The reason is that fairly small group sizes in the early phases of the protocol reduce the number of unnecessary nonempty verification broadcast steps.

We shall now discuss the question of the optimal selection of group sizes in the partitioning protocol. If we choose to divide the partition into N (under the assumption that we know N) sub-partitions with 1 station each, the protocol degenerates. Stations choose a sub-partition from 1 to N, meaning that the stations will broadcast in the next broadcasting step with probability of $1/N$. No other sub-partition will be created until first one is non empty (with as many as necessary number of retries). If a sub-group of only one station is created, an *Id* can be assigned to that unique station. This resembles to the "simple competing protocol" (*HNO-known-N*). However, one major difference still remains in the case of collision while creating such a small sub-partition. The binary dividing protocol would proceed with that newly created sub-partition, while simple competing algorithm would give up and all the stations that were involved into the collision are put back together with the rest of stations to compete for the next *Id*. This simple competing protocol does not have a good performance because of high number of retries introduced by dividing into very small sub-partitions.

It seems that we need to find a compromise for the size of the groups stations are divided into. We are proposing a hybrid protocol that utilizes advantages of both ones. From "outside" the protocol performs as the simple competing protocol, where stations compete for an *Id* with probability p/N. Parameter p is the size of a sub-partition we prefer to get after the partitioning and its best value will be experimentally determined. The expected number is about 2 (the number is equal to $N/2$ for the starting *HNO-unknown-N-mod* protocol). In the case of a silence or unique broadcast the behavior of the protocol remains the same, but in a case of collision, stations do not go to the next iteration where all of them again compete. They can take advantage of the fact that it is small number of them that collided, and can agree about *Id*s faster than the corresponding protocol with all stations competing. Instead, only stations that actually collided continue to compete for an *Id*. While the dividing protocol is being performed, other (unassigned) stations simply listen and wait until all stations in that small group receive their *Id*s

before they can continue the original process of the competition. When the original process of simple competing protocol is re-established, there will be a smaller number of stations left to compete. Therefore, the stations will broadcast with probability $p/(N-P)$, where P is the number of stations that were assigned *Id*s during the just finished dividing protocol. More precisely, we can assume that the probability is p/n, where n (once initialized to N at the very beginning) is constantly updated (recall that pending stations listen to the dividing protocol, and are able to accordingly update n). The nominator is still p, since we again desire to get p-sized group of stations after our partitioning. Potentially, the best value of p could depend on n. However even if that is the case, we anticipate that the impact of variable rather than constant value of p on the overall performance of the algorithm is not significant.

The corresponding protocol is given in pseudo-code and with a higher level description. We do not present detailed version of protocol since it is fairly long and similar to the version of the hybrid protocol where number of stations is not known. Such a protocol is more important one and will be presented in more details in the next section.

```
Protocol Hybrid-known-N

Dividing sub-protocol
repeat
   create nonempty partition1;
   create nonempty partition0;
   if any partition has a single station {
       the station gets assigned Id and becomes passive;
       n ← n - 1; }
   partition1 continues partitioning, partition0 waits;
  if partition1 empty the next previous waiting partititionO continues partitioning;
until no more partition0 waiting;

/* main */
n ← N;
repeat
   unassigned station broadcasts with probability  p/n;
   switch  (status of the cannel) {
     case silence:
    do nothing;
     case single:
        n ← n - 1;
        station that sent the message gets assigned Id and becomes passive;
```

```
        case noise:
            station that collided enters Dividing sub-protocol;
            /* that may take several broadcast steps */
            active station that did not collide just listens and updates n until the sub-
    protocol is finished;
      }; /* end of switch */
    until n = 0;
```

4.4. A hybrid protocol with unknown, but estimated number of stations

The original hybrid protocol, which assumes that the number of stations is known, can be modified to work for the case of unknown N, by adding some estimation techniques. In protocol *Estimated-N* a very passive initial estimation is chosen, in order to avoid separate unnecessary estimation steps at the beginning. The estimation is updated during the course of the protocol. That passive approach is not applicable in this case. Since the protocol starts as a simple competing protocol where station compete for an *Id* with probability p/m (m estimated), it would be very inappropriate to have m estimated at *2* or so. Certainly many stations would collide and consequently start performing dividing protocol which is not our goal, because at that moment a very large number of station would collide, while dividing protocol performs well for small number of stations. Thus, we need first to estimate the number of users, and then to start our hybrid protocol. Unfortunately those steps would be a significant overhead for the case of small number of stations since $\log N$ (which is roughly the number of steps for estimation) is comparable to N for small N. That overhead is insignificant for a large number of users, although we still can expect worse results comparing to *Hybrid-known-N* protocol due to errors of estimated m.

During the execution of the protocol, the estimation of m is constantly updated. The logic of updating mentioned in protocol *Estimated-N* applies here as well. That is, update of m can be incremental (± 1), or more aggressive (exponential, for example). In the formal presentation of our algorithm we do not describe all the details of the updating of m. In order to keep the description simple, we merely state when increase or decrease is to be done. We used exponential update in our simulation analysis. The simple competing protocol would update m for the cases of silence, single broadcast and collision as the original algorithm. In addition, the nested *HNO-unknown-N-mod* would also update m, but only in the case when an *Id* is assigned (and m is therefore decremented).

We shall now describe a simplified version of the protocol, while the more detailed version is presented later by using a finite state machine that shows stages one station can go through during the execution of the protocol.

```
Protocol Hybrid-estimated-N

Dividing sub-protocol
repeat
   create nonempty partition1;
   create nonempty partition0;
   if any partition has a single station {
      the station gets assigned Id and becomes passive;
      m ← m - 1; }
   partition1 continues partitioning, partition0 waits;
  if partition1 empty the next previous waiting partititition0 continues partitioning;
until no more partition0 waiting;

/ * main */
estimate m;
repeat
   unassigned station broadcasts with probability p/m;
   switch  (status of the cannel) {
     case silence:
    decrease m;
     case single:
        m ← m - 1;
        station that sent the message gets assigned Id and becomes passive;
     case noise:
        station that collided enters Dividing sub-protocol;
         active station that did not collide just listens and updates m until the sub-
protocol is finished;
         increase m;
   }; /* end of switch */
until previous estimation was 1, and we got "single";
```

The state diagram actually explains in more detail how these two algorithms were merged. Let us first explain the convention for drawing the state diagram.

- States are presented with circles. The name of the state and an action that is done when the state is left (after "the slash symbol" /) are given inside the circle.
- Some states have sub-states, presented with a smaller circle inside a bigger one. Action specified in state can be overridden by action of the sub-state (if such is specified). If no

action is specified for a sub-state, the action of the state is performed when the sub-state is left.

- Transitions (associated to broadcast steps) are presented with arrows. Each transition is associated with two expressions. First one (before "the slash symbol" /) is a condition for the transition. The condition is evaluated after the action of the leaving state is performed. Conditions from all the transitions leaving one state must be complete (must cover all the cases) and must not be in a contradiction. The second expression (after "/") is an action that is performed prior to entering the next state. Note that these actions can be very complex and may be of conditional nature, e.g. with if statements. Those conditional expressions should not be mixed with transitional condition.
- When a station changes a state (through a transition) that has sub-states, the sub-state remains the same unless it is specified differently. In such case a transitional condition and a transitional action are also given, which are specific to that sub-state transition.

Now we are able to explain the presented state diagram. Although the state diagram is much more detailed than the pseudo algorithm above, its still lacks some subtle details. They are omitted again for the sake of the simplicity of the graph.

A station enters first the estimation state with no transition condition, initializing m with value 2. While leaving the state, it broadcasts with probability ½. It remains in the same state as long as a collision happens (transitions condition is "$recv = 2$", which means that the station, after the broadcast step, has received 2 or more messages). In that transition m gets doubled. In case of a silence after the broadcast step, the estimation is finished and station enters "competing protocol".

Competing protocol consists of two states: active state and passive state. A station in active state did not get *Id* assigned yet and actively participates in the protocol. Once a station gets assigned *Id* (transition from active to passive with condition "$recv = 1$, *sent*", that is, a single broadcast and this station actually sent the message) it goes into a passive state. That means, the actions while leaving active and passive states are different. While leaving active state, the station broadcasts with probability p/m, whereas it does not broadcast upon leaving passive state. These two states are very similar from other perspectives. That is, the passive and active stations that did not get assigned *Id* merely update m accordingly. Note that a station may

enter competing protocol from a passive rather than active state, but only in the case when the estimation protocol finishes with a single broadcast instead of a silence. The station that actually sent that message will go to a passive state, and all other stations will start competing protocol from the active state.

In the event of a collision, (during the competing protocol), each station (passive or active) enters "dividing protocol". It is not so obvious to comprehend why passive stations should enter dividing protocol as well. However, this enables passive stations to monitor active stations, update necessary parameters, and accordingly learn when the dividing protocol has finished.

The dividing protocol itself is represented with two states: *partition1* and *partition0.* Sub-states are introduced since both active and passive stations enter the dividing protocol, and they have different behavior. Sub-states are also needed when stations return from dividing to competing protocol, as a piece of memory needed to remember where the station came from. There are three sub-states: active sub-state, passive sub-state 1, and passive sub-state 2. Stations from active sub-state are those that did not get *Id* assigned yet, and they enter dividing protocol with clear intention to get one. They left active state from competing protocol since they collided between themselves. Stations from passive sub-state 1 came also from active state of competing protocol. Such stations also do not have *Id* assigned, and do not actively participate (do not send a message) since they were silent when they left active state of the competing protocol. The third sub-state, passive sub-state 2, represents the stations that came from passive state of competing protocol. They have their *Id* assigned, but still need to follow the course of the dividing protocol to be able to know when the competing protocol reestablishes.

While in dividing protocol, a station may change sub-state under particular condition. A station moves from active sub-state to passive sub-state 2 after receiving its *Id.*

When stations leave dividing protocol end reestablish competing protocol, they enter the state (active or passive) according to their sub-states. That is the same state they were in when they entered dividing protocol, for the most of the stations. Exceptions are, of course, the stations that change the sub-state. Since stations that entered active sub-state (coming from active state of competing protocol) go to passive sub-state 2 (which represents the stations coming from the passive state of competing protocol) when they get assigned *Id,* those stations will actually

reestablish the competing algorithm in the passive state. The reason is that the same transition from active to passive state in competing protocol means that a station got assigned its *Id.,* even though such a transition occurred indirectly.

Let us give more explanation for transitions in dividing protocol. Starting state for dividing protocol is state *partition1*. There is no action associated to that state, but they are defined at sub-state level. In fact, only one sub-state has an action, namely the active sub-state. The action in the active sub-state is as follows. A station may leave the sub-state (and consequently the state), but only at the deepest level of dividing algorithm (condition $l=L$), and broadcast with probability ½ (which is the essence of the dividing protocol). There are three transitions going from state *partition1*. In the event of a silence ("$status0 = 0$"), no action is performed and station remains in the same state (and sub-state), because an unsuccessful attempt to generate a non-empty partition just occurred. In case of a collision, a partition is successfully created, and thus the next step is to check whether remaining partition is non-empty. Therefore, there is no action, and station moves to state *partition0* (to the same sub-state). If a single broadcast occurred, a non-empty partition is created similarly to the case of a collision. Consequently, the station enters *partition0* state and decrements *m*. Only one station has overridden behavior, which is the active station that broadcast (transition condition "$recv = 1$, *sent*", moving from active sub-state; it sent a message while leaving *partition1*). It also joins *partition0,* and decrements *m*, but also gets assigned *Id.*

The protocol that divides stations into two groups, *partition0* and *partition1,* and makes decisions according to *status0* and *status1,* is similar to the one described for the protocol *HNO-unknown-N.* There is an overridden behavior for active sub-states in *partition0* state, when an active station broadcast, and it happens to be a unique broadcast. In that case the station does change the sub-state to passive 2. Another overridden behavior for *partition0* appears when dividing protocol finishes. While leaving *partition0* state, if *L* was decremented and reaches 0, then there are no more pending stations in dividing algorithm. Thus stations shall return to competing algorithm and there are two transitions that show it. There is no such a transition from active sub-state, because there is no station that is supposed to be in that sub-state at that moment (all of active ones went to passive 2). There is a 'hidden' detail in the just mentioned transition. If during the course of the algorithm our estimation for the number of unassigned stations reaches 0 or less, we need to bust it to the value at least *1*.

It remains to explain when the stations leave the main protocol. It is always done from competing portion of the protocol, more precisely from passive state, since there are no stations supposed to be in active state at that time. The criterion is the same as for *Estimated-N* protocol. A parameter *prevM* is used to describe what was the value of m in the previous broadcast step (another 'hidden' detail). If that previous estimation was 1, and current m is less than 1 then our previous estimation was correct, and thus all the stations (and all of them are in passive state) are ready to leave the protocol and start using assigned *Id*s.

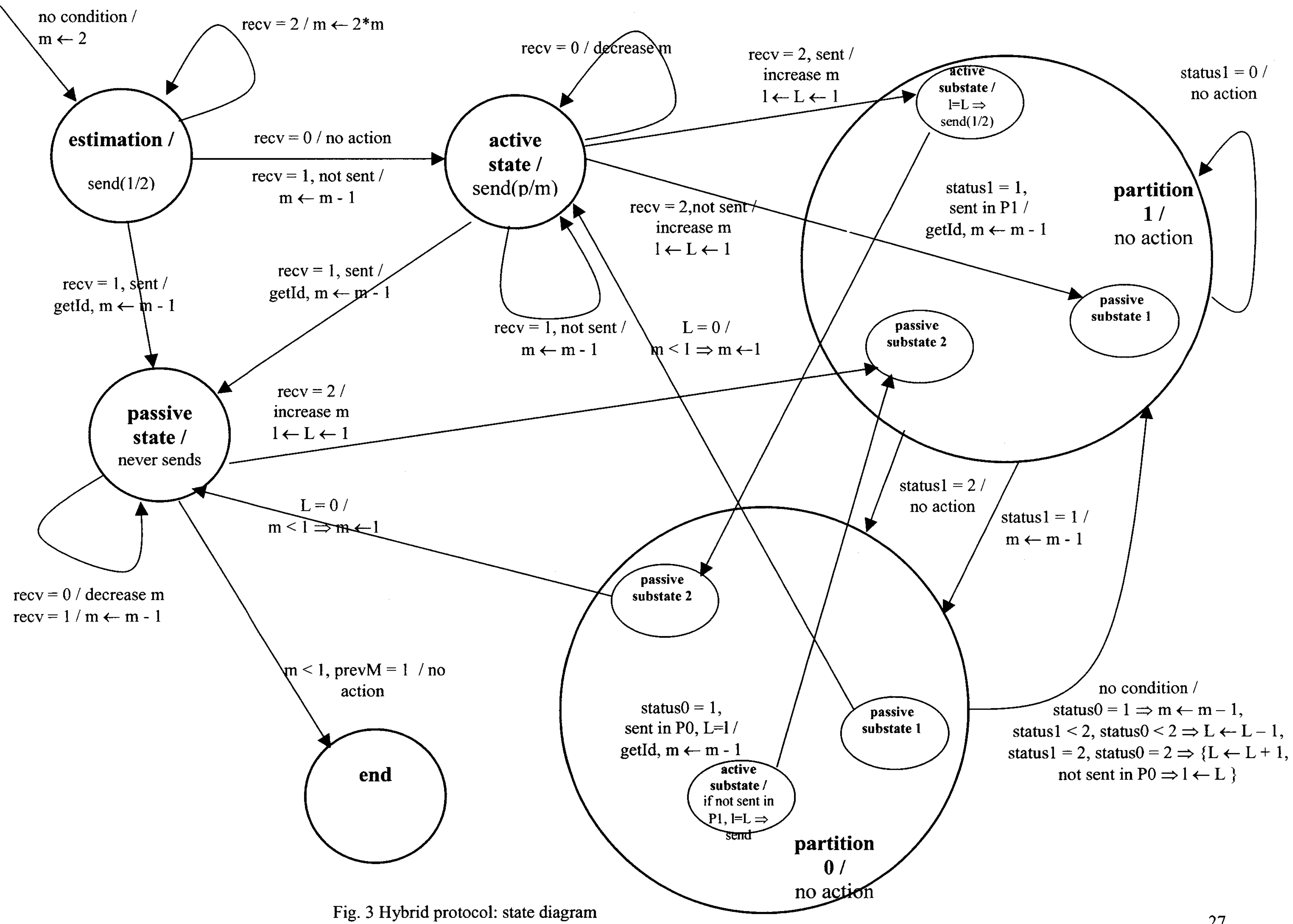

Fig. 3 Hybrid protocol: state diagram

5. Deterministic initialization protocols

We shall now study a new class of initialization protocols. Each station is assumed to have a distinct serial number that will be used in as a source of quasi-random generator, and thus the sequence of random numbers generated would be always the same. The initialization protocols are then deterministic. That is, whenever the protocol is performed among the same set of stations, each station in the set will get the same *Id*. Instead of choosing between 0 and 1 at random, the stations use one of their bits from the serial number to compete for *Id*s.

5.1. A deterministic version of *HNO-unknown-N-mod* protocol

It is relatively straightforward to change *HNO-unknown-N-mod* protocol to be deterministic. Because of similarity, we will not consider here the deterministic variant of the known-number-of-station protocols. We shall concentrate here only on the more generic unknown-number-of-station protocols. The decision whether to select 0 or 1 in the protocol is not taken at random with probability of ½ but instead depends of the bit value at appropriate position of serial number (in binary format). More precisely, the position where the bit is queried is the first not yet used position. Thus while stations in a group have the same sequence of 0s and 1s at first k bits, they belong to the same partition. These stations will try to partition themselves using *(k+1)-th* bit: all stations that have bit 1 at that position will broadcast in *partition1* and will remain in the same group since they now have first $k+1$ bits matching. The remaining stations will broadcast in *partition0.* If any of the partitions is empty, stations remain in the same partition, and the process continues by using bits at position $k+2$, $k+3$... as criteria for partitioning. Clearly, for a set of distinct serial numbers, k cannot exceed the length of the serial number.

The deterministic version of the protocol closely resembles its original protocol. We refer to it as the protocol *HNO-det,* where *unknown-n* is implicitly understood. We introduce a variable *bitPosition* that is an index to the current bit used for partitioning criteria. It is incremented each time a new iteration of partitioning is entered. We refer to iteration as a successfully complete or incomplete pair of broadcast steps for generation of two non-empty partitions. There are several different cases on how to enter new iteration of partitioning (i.e. unsuccessful creation of non-empty *partition1*, successful creation of non-empty *partition1* with

unsuccessful creation of non-empty *partition0* etc.). In each case, a new iteration starts with an attempt to generate *partition1*. Consequently, the variable *bitPosition* is incremented inside the loop that is responsible for creation of *partition1,* slightly prior the broadcast step itself.

Only stations from the current partition will increment the *bitPosition.* These stations are active stations (they did not assign their *Id* yet), at the deepest level of the partitioning ($l=L$). Clearly, the value of *bitPosition* is irrelevant for passive stations, since they already assigned their *Id*s; nevertheless they still need to listen and update parameter L in order to learn when the initialization protocol finished. Stations that are active, but are not at the highest level of partitioning ($l<L$) do not increment *bitPosition*, because they are in the pending state. They have already gone through l successful partitioning iterations and their first *bitPosition* (which is $\geq l$) bits match the corresponding bits in active stations. These stations continue the competition for *Id* when parameter L is decremented up to the value of their local value l. Stations then reestablish partitioning, and continue incrementing *bitPosition.*

The deterministic protocol is expected to have similar performance as the corresponding randomized one, assuming a 'randomized' distribution of 0s and 1s in the binary representation of stations' serial numbers.

<u>Protocol HNO-det</u>

```
l ← L ← nextId ← bitPosition ← 0;
SerialNum[LENGTH]; // binary representation of ser. number from 1 to LENGTH
station active;
repeat
   active station that has l < L just listens and updates nextId and L;
   repeat
      active station: bitPosition++;
      if (active station and SerialNum[bitPosition] = 1) then
        station broadcasts;
      let Status1 be the resulting status on the channel;
   until Status1 <> 0;
   if Status1=1 then {
      only station that sent in partition1: Id ← nextId and becomes passive;
      nextId ← nextId + 1 };
   active station that didn't send in partition1 broadcasts;
   let Status0 be the resulting status on the channel;
   if Status0 = 1 then {
       only station that sent in partition0: Id ← nextId and becomes passive;
```

```
        nextId ← nextId + 1};
    if Status1 < 2 and Status0 < 2 then { L ← L - 1};
    if Status1 = 2 and Status0 = 2 then {
        L ← L + 1;
        active station in partition1: l ← L };
until L=0;
```

5.2. A tree-based deterministic protocol

We shall now propose a new deterministic protocol, called *Tree-det.* As in the other deterministic protocols, the criteria for assigning an *Id* is based on the value of the unique serial number (e.g. 32-bit number). Stations compete between themselves for *Id*s by comparing the bits in their serial numbers. In any competition round, only (active) stations with bit 1 broadcast at a particular position in the serial number broadcast, and we say that they beat those (active) stations having bit 0 at the same position. The competition starts from the most significant bit. Defeated stations wait (and remember their defeat level), while the remaining stations continue competing, using the next bit position. The process continues until the 'winning' group (of stations with 1 at appropriate bit positions) becomes a single station group. After that, the defeated group with the highest defeat level reestablishes process of fighting.

Although similar, the protocol has a major difference compared to the protocol *HNO-det.* The defeated stations (which could be associated to *partition0*) do not broadcast in the subsequent broadcast step to ensure that there is a non-zero number of them. Protocol *Tree-det* ignores (more precisely, avoids) this verification step, with the same motivation as in hybrid unknown-*n* type of protocols. We do not employ the idea of hybrid protocol here, but we still need a mechanism that will compensate for that skipped verification (whether or not the set of defeated stations is empty). One such mechanism is indeed necessary to decide what is the point (that is, bit position) where the defeated stations will reestablish the process of competing for an *Id.*

The simplest way of finding the deepest level of non-empty defeated set of stations is to search for it in a backtrack fashion. While stations advance in their process of competing, the bit position used as a criterion is incremented. The searching process begins after a single station is assigned its *Id.* The stations then start reverse process of decrementing bit position, in order to find the deepest level of defeated stations (each station remembers its level of defeat). The bit

position is decremented until a non-empty group of stations with given defeat level is encountered. More precisely, stations will not reestablish the competition process until the competition level (bit that decides on whether or not to compete) is set back at a level before the defeat level, while the stations were still winning.

In summary, the new approach does not verify whether set of defeated station is nonempty, but instead introduces new broadcast steps in reverse process of searching for the deepest level of defeated stations. Since it is relatively complex to analytically compare these two approaches, we implemented them and compared experimentally. *Tree-det* protocol can be described as follows:

```
Protocol Three-det

lastId ← 0;
L ← 1; /* competition level */
/* a(b) - array representing serial number in binary form */
/* d – defeat level */
sign ← +1;  /* direction of three search */
station becomes active;
repeat
   active station with a(L)=1 sends a message;
   switch (status of channel)
      case noise:
         active stations with a(L)=0 become passive, d←L;
         sign ← +1;
         L ←L + 1;
      case single:
         lastId ←  lastId + 1;
         station that sent the message gets ID ← lastId, becomes passive, d←null ;
         sign ← –1;
         L ←L – 1;
      case silent:
         L ←L + sign;
    endswitch;
    if d=L then
         the station becomes active;
until L = b+1;
```

Variable L is used as a pointer to the currently used bit in a serial number (similarly to variable *bitPosition* in protocol *HNO-det*). All the stations, active and passive, always update variable L (this is different than corresponding action in protocol *HNO-det*), since it is used as a

criterion to exit the protocol. Variable *sign* indicates the current direction (increasing or decreasing) and is used to accordingly update variable L. Positive *sign* indicates that stations are advancing in competing process, while negative one means that stations are in reverse process (searching for the deepest level of defeated stations). Another variable introduced is defeat level d. It is a local variable, i.e. each station has potentially different value for the variable). It is initialized whenever a station is defeated in the process of fighting and is used to convert a station from passive to active whenever variable L reaches its value during the reverse-search process. Another difference from protocols *HNO-det* (at least in terms of terminology) is that passive stations in protocol *Tree-det* is not only stations that got *Id* assigned, but also defeated ones (and thus waiting for their *Id*s).

Once a group of stations broadcast, protocol distinguishes three different cases:

- In case of a collision, regardless of the stage the stations were in the previous step, the competition process proceeds. Thus, *sign* is set to $+1$, and L is incremented.
- In case of a single broadcast, regardless of the stage the stations were in the previous step, stations start searching for the deepest level of defeat. Thus, *sign* becomes -1, and L is decremented. The station that broadcast that unique message assigns its *Id*, becomes passive, and its variable d assumes some dummy value to prevent mixup of passive defeated stations and passive stations with assigned *Id*.
- In case of a silence, stations proceed with the process they are currently in. Thus, L is updated with the value *sign*. Silence occurred because all the active stations had 0 at the current bit. If that event occurred during the competition process, no station is winner in that competition step. Therefore the next bit position should be taken in the next broadcast step, meaning that stations keep updating bit position for future competition in the same direction. If silence appeared during reverse-search process, stations continue in the same direction, because they need to discover the bit positions of candidate stations while they were still winners, which is a bit position where they have value 1. A station reestablishes the competition process when it becomes active during the reverse-search process, which occurs when L reaches station's defeat level d.

We will trace the algorithm on an example. Consider 8 stations (from A to H), with 11 bit long serial numbers (bit at position 1 is the most significant bit):

1	2	3	4	5	6	7	8	9	10	11	
1	0	1	0	1	0	1	0	1	1	1	A
1	0	1	0	1	0	1	0	0	1	0	B
1	0	1	0	0	1	1	1	0	1	1	C
1	0	1	0	0	1	0	0	1	0	1	D
0	0	1	1	0	1	0	1	1	1	1	E
0	0	1	1	0	0	1	0	1	0	0	F
0	0	1	1	0	0	0	1	0	1	0	G
0	0	0	0	1	0	1	1	0	1	1	H

Table 1. Tree-det: serial numbers

The nodes of the graph represent broadcast steps and stages the stations go through. Arrows represent transitions of node from one stage to another after a broadcast step. The depth of a graph node is equal to the bit level that is used in the broadcast step represented by that node: if it is L-deep node (starting from 1 as root node) of the graph, the L-th bit is used for that broadcast step. That is true for both directions of bit level change. If bit level change is positive (i.e. stations are in the process of competing), stations advance to a new stage, represented by a new node. While bit level change is negative (i.e. stations are in reverse-search process), stations do not advance to new stages, but instead move back into formerly passed stages. The number of arrows going out of a node ranges from 1 to the number of stations that are active in the stage represented by that node. This means that the same number of arrows will be coming into that node, since that many times stations will come back to that bit level position. With each new returning arrow, the number of active stations will decrease. Therefore one node represents more than one stage of the algorithm (however, those stages are at the same bit level). In order to distinguish them, stages are indexed. When a node is visited for the first time, it is indexed with number 1, for the second time with number 2 etc. For each of these stages, the list of active stations, and the list of stations that will broadcast in the next broadcasting step, at that node is also included.

For example node 14 is visited three times. First time it is visited, stations *E, F, G* and *H* are active, but only *E, F* and *G* broadcast at the next broadcast step (since only those three out of 4 active ones have bit 1 at bit position 3). After the broadcast step station go to new stage represented by node 15, only *E, F* and *G* are active at that new node, while station *H* becomes defeated. The second time node 14 is visited, only station *H* is active (newly active; it was

previously passive, with its bit level being exactly its defeat level). No station broadcast in the next step (symbol ∅).

All the stations enter the protocol with initial stage represented by node 0. All the stations are active at that time and only those with bit 1 at bit position 1 broadcast. In this example those are stations *A, B, C* and *D.* They are the only ones to remain active in the next broadcast step, while stations *E, F, G, H* become defeated. Stations *A, B, C* and *D* remain active until node 4, since meanwhile all of them either have 0 or 1 at appropriate bit level. At node 4, only stations *A* and *B* broadcast and remain active until node 8, since both of them have either 0 or 1 at appropriate bit level. At node 8 (bit level 9) only node *A* broadcasts, gets assigned *Id,* and directions is changed, thus the next bit level position is one less (8). Station *B* remained active (it could be considered that *B* was temporarily defeated, but its active state is immediately restored). Bit level *L* continues to be decremented until one of active stations broadcast 1, which means that the deepest level of defeated stations is found. Since there is a unique such station *B,* there is no need to re-establish competing process yet. Station *B* then assigns *Id,* becomes passive, and stations continue reverse-search process. Stations *C* and *D* become active at node 4, but competing process is re-established at node 2, which is the deepest position where active station had 1 at current bit position. The new competition round is represented by a new branch of the graph, and the process resembles the competition process of stations *A* and *B.* When the process finish, stations return to node 2 again, for the third time. Since no active station remains, they proceed to node 0, where stations *E, F, G* and *H* restore their active states. Although they do not broadcast 0 at that stage, they re-establish competing process, since they cannot go further in reverse-search process. During the new process, stations *E, F* and *G* receive their *Id*s, while *H* remains unassigned. Thus a new competing process re-establishes, and the last active station *H* gets assigned *Id.* Since stations do not know their number, they return again to node 0 and re-establish a new competing process for the forth time for this node. That fighting process is the last one since all the stations reached the end of their serial number, meaning that all stations received their *Id*s.

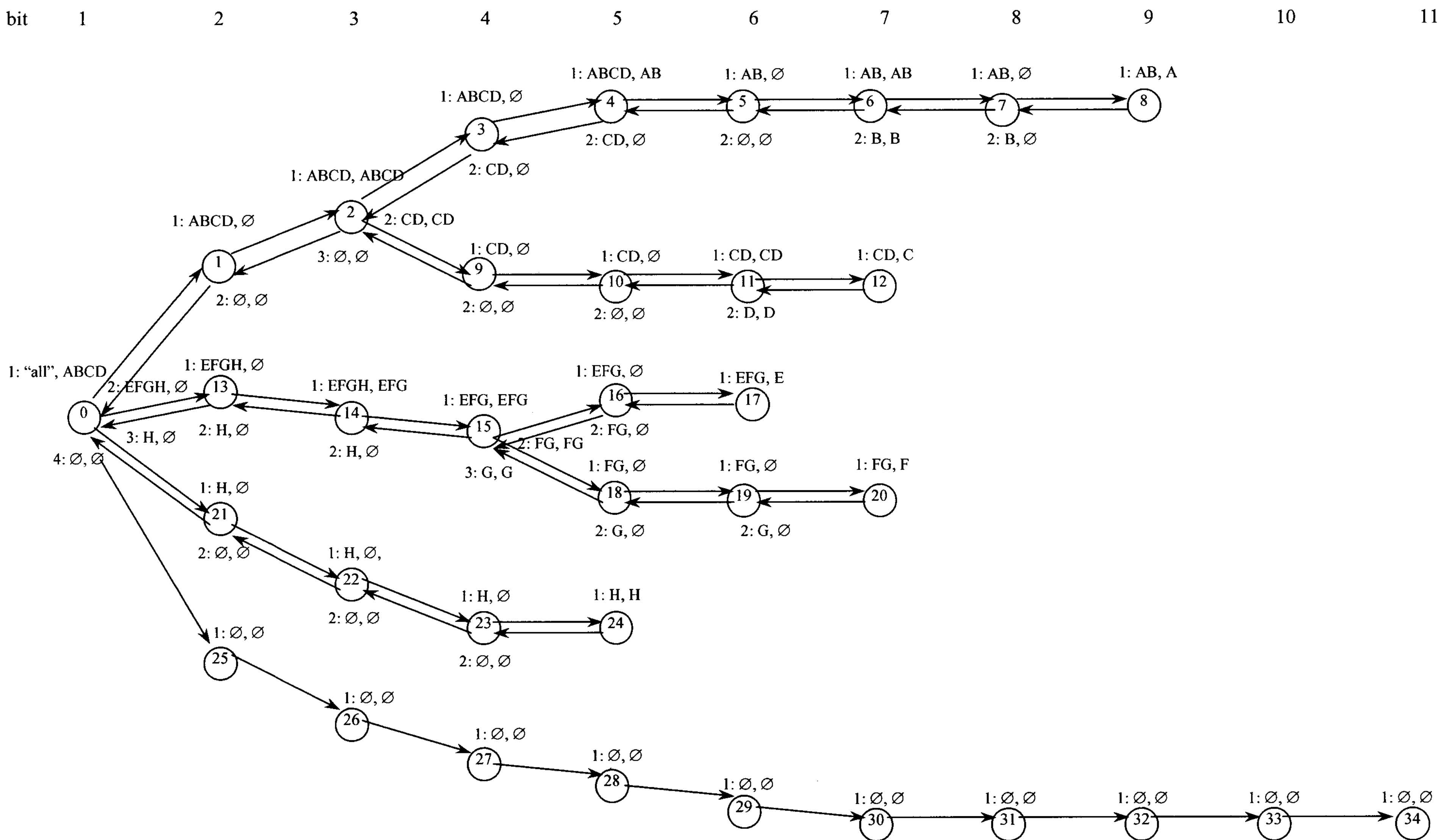

Fig. 4 Tree-det: trace example - original

This algorithm can be further improved. The improvement will be first explained by using the next graph, for the same example. A closer look at the example reveals that stations *C* and *D* were active in nodes 2, 3 and 4 (at bit positions 3, 4 and 5), and became passive in node 5. After that, they remained passive until all the active nodes (*A* and *B*) received their *Id*s. After the competing process was reestablished for them (in node 2), they continued to compare bit position 4 and 5 (node 9 and 10). But they already compared those bits in node 3 and 4, and are aware that these bit positions of other stations did not change when the process reestablished. Thus instead of incremental move forward, a kind of advanced forwarding can be implemented. In our example, this means that, after stations *C* and *D* re-established process at bit level 3, they can directly compare bits at level 6 (a direct arrow is shown from node 2 to node 11 at the second graph). Other examples of that optimization can be seen in direct arrows from node 15 to node 20, from node 0 to node 23 and from node 0 to node 29. The only case in our example when such optimization cannot be applied when competing process is re-established, is the second broadcast in node 0. Since actives stations at that node, *E, F, G,* and *H* where not active in node 1, they do knot know what they have at bit position 2. So they have incrementally to continue with fighting process.

Another similar improvement of the protocol can be made in the process of backward search. Consider nodes 6, 7 and 8. While going backward, no broadcast step is needed at bit position 8 (node 7), since all the defeated stations at node 8 (station *B*) already had a chance to broadcast at bit position 8 during the competition process. They know that none of them will broadcast at position 8, and thus they can compare bits at position 7, which is the deepest bit position that had collision in the just finished competing process. There are few more instances for this optimization in our example. However, even with this optimization, it may take more than one step for reverse-search process in order to re-establish competing process (for instance, consider arrows from 6 to 4 and from 4 to 2).

We will describe one more type of optimization that can be applied, and will also explain it using the same example, and show it on the third graph. Consider node 0, which is visited 4 times. No station broadcast occurred at bit position 1 when the node was visited for the second time. This means that, if ever in future active stations visit that node (bit position) again, no station will broadcast. Thus the stations can directly compare the first next bit position. That

optimization can be applied during the whole course of the algorithm, that is, whenever unassigned stations discover a leading 0 in their serial numbers, the 0s can be ignored. For this particular example that leading 0 is discovered at bit positions 1, 2, 3, 4 and 5 at nodes 0, 13, 14 (for second time), 23 and 24.

The third depicted graph contains all the described optimizations. Obviously, number of broadcast slots has been significantly reduced. The number of 'saved' broadcast steps is equal to the number of skipped nodes during the graph traversal. The algorithm is given in source code form using C++. Only essential part of the algorithm is given, describing class *Station*: its initialization, and two function members *clockUp* and *clockDown*. All the stations first initialize themselves, and enter infinite loop where first all of them perform *clockUp* what is a simulation of the broadcast itself, and all of them perform *clockDown* what is all the necessary computation in the algorithm based on just performed broadcast result. The above described optimizations of the algorithm are done through stack data structures *stackNextForward* and *stackNextBackward*.

```
Station::Station() {        // constructor: initializes a station
  serialNum = pSerialNumServer->getUniqueSerialNum();
  nextId = 0; L = 1; myId = 0;
  active = 1;    // True
  receiveChannel = 1;
  numOfMessagesReceived = 0;
  dataReceived = 0;
  defeatLevel = 0;
  sign = 1;
  spNextForward = -1;
  spNextBackward = -1;
}

void Station::clockUp() {

    if (!active) return;
    bit = serialNum.getBit(L);
    if (bit) theEther.messageBroadcasted(1, 0); // channel 1, content not important (0)

} // end of Station::clockUp()

void Station::clockDown() {

 switch (numOfMessagesReceived) {
        case 0:
        if (sign > 0)
          L = L + 1;
        else {
          if (spNextBackward >= 0)
            L = stackNextBackward[spNextBackward--];
```

```
      else {
        sign = 1;
        L = L + 1;
      }
      if (spNextForward >= 0)
        spNextForward--;
    }
    break;

  case 1:
    nextId = nextId + 1;
    if (bit) {
      myId = nextId;
      active = 0;
      defeatLevel = -1;
      bit = 0;
      counterDone--;
    }
      if (sign == 1)
        stackNextForward[++spNextForward] = L;
      else
        spNextForward--;
    if (spNextBackward >= 0) {
      L = stackNextBackward[spNextBackward--];
      sign = -1;
    } else {
      L = L + 1;
      sign = 1;
    }
    break;

  default:
    if (active && !bit) {
      active = 0;
      defeatLevel = L;
    }

    stackNextBackward[++spNextBackward] = L;

    if (sign == -1) {
      if (spNextForward >= 0) {
        L = stackNextForward[spNextForward--] + 1;
      } else
        L = L + 1;
    } else {
      stackNextForward[++spNextForward] = L;
      L = L + 1;
    }

    sign = 1;
    break;
} // end of switch statement

if (defeatLevel == L) {
  defeatLevel = 0;
  active = 1;
```

```
    }
    if (L > lengthOfSerialNum) {
     if (myId == 0) {
       counterDone--;
       myId = ++nextId;
     }
     counterDone--;
    }
} // end of Station::clockDown
```

There is one obvious disadvantage of this protocol. It ends when we go through all the bit positions, and having very long serial numbers (for example 64 bits), even if only a smaller portion of the serial number is needed to resolve all the *Id*s, an additional approximately constant overhead is added. As we mentioned at the beginning of this section of deterministic protocol, a simplified version if it may be used in case of known number of users. The additional information could then be easily used as a criterion to exit the protocol, and thus completely reduce just mentioned overhead. That overhead is indicated by the last sequence of competition rounds which does not produce any new *Id*s.

Fig. 5 Tree-det: trace example - optimization 1

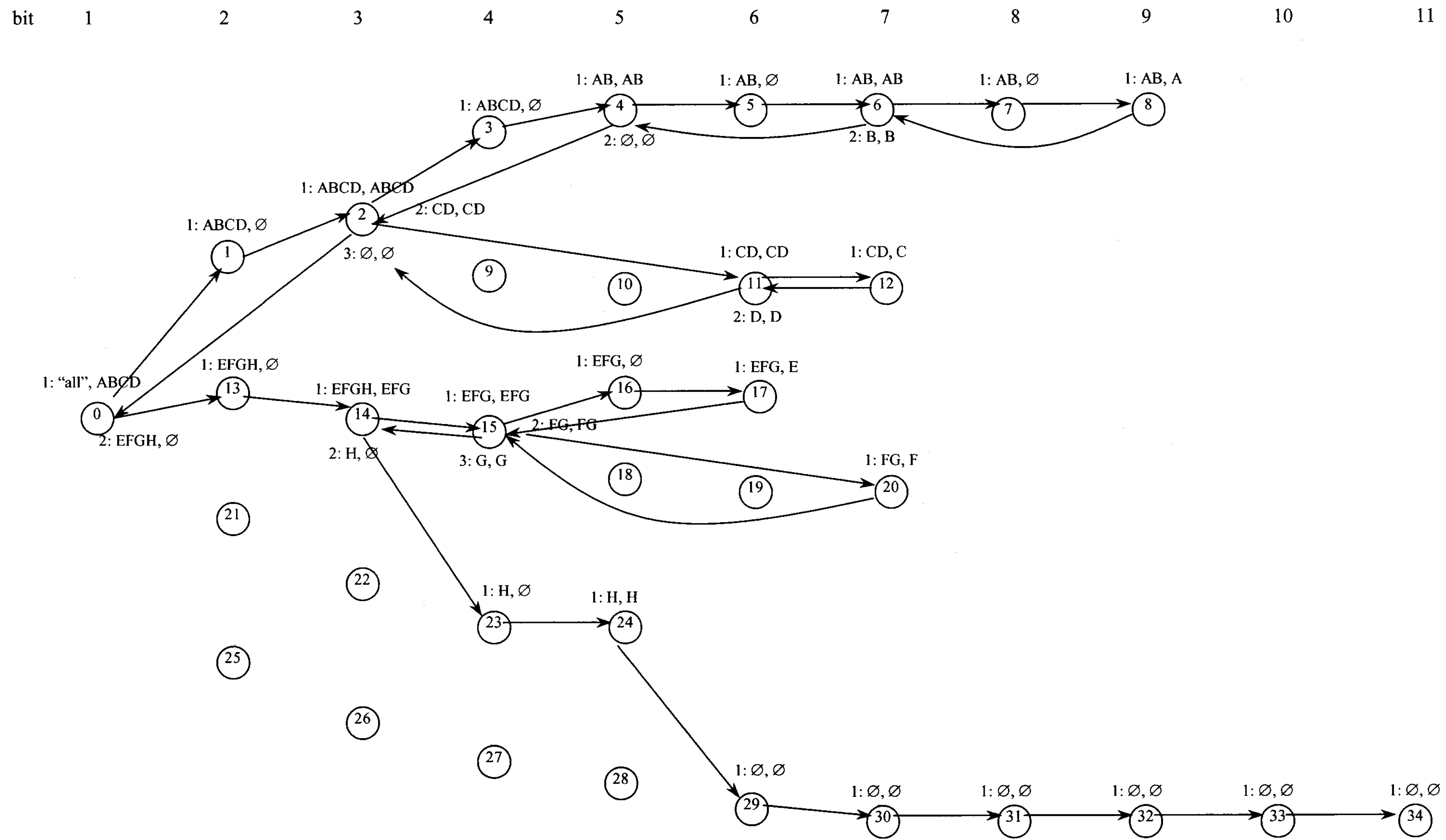

Fig. 6 Tree-det: trace example - optimization 2

6. Performance evaluation

We shall now present simulation results for the described initialization protocols. All the protocols are simulated at a single node computer (although the protocols are, in its nature, distributed). Each station is a finite state machine, where state changes (according the rules of a protocol) after each broadcast slot. Thus, the performance is measured as a number of broadcast slots needed to complete the protocol, where any computation time is neglected. Each protocol is simulated with different number of stations: 3, 10, 30, 100 and 300. For accuracy, each experiment is repeated 100 times.

6.1. Deterministic

# of stations	3	10	30	100	300
ser. # length			***Tree-det***		
2	3.18				
4	6.04	16.18			
8	10.59	24.97	69.51	211.97	
16	19.00	33.34	83.25	268.02	796.07
32	35.57	49.56	101.32	290.08	811.99
64	66.78	83.35	136.12	328.60	844.56
			HNO-det		
2	4.00				
4	5.15	18.77			
8	6.13	23.49	70.83	217.04	
16	6.03	23.75	76.92	264.77	795.82
32	7.50	23.82	78.88	270.93	795.57
64	5.73	25.49	81.18	276.83	795.32

Table 2. Performance analysis: deterministic

Table 2 summarizes the performance of deterministic initialization protocols. The performance is measured only for the case of unknown number of stations. There are two protocols in this category: *HNO-det* (our modification of *HNO-unknown-N*) and the proposed original *Tree-det*.

HNO-det protocol outperforms *Tree-det* in majority of test cases. This is due to the already explained overhead introduced with long serial numbers in *Tree-det* protocol. The difference in performance between the two protocols is less than (but close to) the length of the

serial number. For short serial numbers, *Tree-det* protocol shows insignificantly better results. Experimental results show that for a given number of stations 2^k and k-bit serial numbers, the number of broadcast steps for *Tree-det* and *HNO-det* are approximately $2^{k+1}-2-k$ and $2^{k+1}-2$, respectively.

6.2. Random-known

# of stations	3	10	30	100	300
HNO-known-N					
	5.19	22.61	74.54	264.18	800.66
***Hybrid-known-N* (p=1)**					
	6.02	20.18	62.67	215.98	654.31

Table 3. Performance analysis: random-known

We now compare two non-deterministic initialization protocols for the case of known number of users: the simple competing protocol *HNO-known-N* [HNO] and our hybrid protocol. Although the hybrid one introduces an overhead with impact on performance for small number of stations, it clearly outperforms the simple competing protocol. The complexities of the protocols appear to be $e \cdot N$ and $2.2 \cdot N$.

6.3. Impact of parameter *p* on *Hybrid-known-N*

***Hybrid-known-N* (*N* = 30)**								
p	0.8	1.0	1.2	1.5	2.0	4.0	8.0	15.0
Step #	65.62	62.67	63.52	64.33	66.93	73.51	76.57	78.9

Table 4. Performance analysis: parameter p in *Hybrid-known-N*

We have measured the impact of the selected value of parameter p (equal to the size of each group) on the performance of hybrid protocols. We tried several values for $p \leq N/2$ ($p=N/2$ corresponds to the partitioning into two groups, exactly like in the original *HNO-known-n* protocol). We "proved" that hybrid protocol performs best when $p=1$. In this case, the hybrid protocol performs (from "outside") as a simple competing protocol, when stations fight for an *Id* with probability $1/N$.

6.4. Random-unknown

# of stations	3	10	30	100	300
Estimated-N					
	7.40	28.76	88.68	290.06	850.51
HNO-unknown-N-mod					
	5.35	21.80	74.71	267.64	801.83
Hybrid-estimated-N					
	8.17	25.79	73.47	232.42	683.56

Table 5. Performance analysis: random-unknown

Perhaps the most realistic scenario is the case of initializing stations with unknown number of them using random initialization protocols. We compared three protocols: two modifications of already proposed protocol (*Estimated-N* and *HNO-unknown-N-mod) and* the original *Hybrid-estimated-N* one. We do not simulate the original protocol *HNO-unknown-N* [HNO], for which authors claim $8 \cdot n$ complexity. As expected *HNO-unknown-N-mod* performs with approximate complexity $e \cdot N$, *Estimated-N* performs slightly worse (due to errors caused by uncertainty of estimated m), and *Hybrid-estimated-N* somewhat better (except for small number of stations where the estimation overhead becomes significant). Protocol *Hybrid-estimated-N* also performs slightly worse than its known-N peer due to errors of estimation of number of stations.

7. Dynamic assignments of channels

After receiving their *id*s, stations are ready to start using their corresponding slots. However, as discussed at the beginning, such fixed assignment of slots is inefficient. In order to organize a dynamic assignment of slots, a communication media/channel has to be reserved for stations to negotiate the distribution of the slots. Commonly accepted idea and practically possible to implement is to use minislots. Each slot with the duration of a fixed sized packet is preceded with a much shorter minislot that would be used for a station to announce (broadcast) necessary information about bandwidth demands for the near future (at least for the coming slot). The assigned minislot guarantees collision free announcement message to the station. Depending on exact protocol, there could be several types of announcement messages. The most obvious one is that a station will inform if there is anything to be sent in the following slot. If not, another station could use the slot. We consider several scenarios how such a minislot could be used to get better utilization of slots themselves.

7.1. Station leaving with notice

If a station is near termination, or does not have anything to send for a longer period of time, it can leave the group. The station would send a special message in its minislot informing all the stations that once the current periodic frame (with N slots) finishes, the following periodic frames will contain $N-1$ slots. When the stations receive that message, they would determine whether their position in the periodic frame was before or after the station leaving. If necessary (for the stations after the one leaving) a station would recalculate its *id* (decrementing it by 1). All the stations will also adjust the local copy of the number of stations.

7.2. Station leaving without notice

Minislots would regularly be used to announce the number of messages in the transmission queue. The station that does not announce such information for a number of consecutive minislots is considered as being inactive, and the remaining stations would implicitly exclude the station's slots from the periodic frame. To distinguish cases of being inactive and having nothing to send, a station would always send a message, even if it does not have anything to send. It may be the very same message that is used to announce that the station

wants something to send. The message would have one parameter indicating the number of messages that are pending in the queue, even being 0. On the other hand, stations that were inactive for a specified amount of time, should recognize the loss of their minislot, and, if they wish to join the group, should proceed as they were new stations.

7.3. New station arriving

Since all the minislots are assigned to active stations, any new station that wants to join the group needs additional bandwidth to announce its arrival. A special minislot in the periodic frame can be used for the station to announce its arrival in the group. Once the active stations learn about the arrival of a new station, they allocate one new slot in the next periodic frame. However, there are some details to be clarified.

A station that wants to join the group has to synchronize with the group. In order to send its request to join the group, it needs to learn where is the empty minislot dedicated for sending such a request. Moreover, it has to synchronize with the whole periodic frame structure. In order to solve this problem, one of the stations from the group (for instance the station with *id* number 1) may broadcast a special framing code (FR) at the beginning of the frame. The new station would then be able to learn what is the period of the frame and how many stations there are in the group (we assume that all the stations have synchronized clock and have the same length of the slot). A special minislot E would be dedicated for new stations. A new station would listen for the traffic and, after recognizing FR, will know that the empty minislot follows and the request could be sent. In the simplest case, the newly arrived station will always get the next available *id* (that is, the current number of stations plus one), and consequently the latest slot (and corresponding minislot) in the subsequent frames.

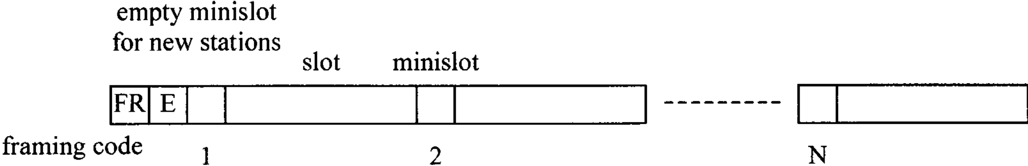

Fig 7. Periodic frame structure

This simple procedure for joining the group works if at most one station uses the assigned minislot to announce its presence and intention. If more than one station attempts to join the group by broadcasting in the assigned minislot, a collision will occur and none of them will join successfully. In such case, the station from the group that is responsible for generation special FR code can decide to offer a "warm welcome" to the new stations. Instead of FR code, another code (say INIT) would be broadcasted. The INIT code indicates that, at the end of the current frame, the stations, members of the group, will stop sending data, and allow the initialization process to proceed instead. The initialization process will finish faster if only new stations participate in it. Alternatively, the initializing protocol may include all the stations, especially if the lower *id* number gives some broadcasting advantage in the protocol. In both cases, one of the initialization algorithms described in this thesis (which assumes unknown number of users) may be applied.

7.4. Channel borrowing

As already discussed, every station has its own minislot and uses it to report the number of packets that it has to broadcast. With respect to slot assignment, there are two possible approaches. In the first one, slots are not pre-assigned to the stations, and a protocol exists to determine which station is to use the next slot. In the second approach, one slot is assigned to each station. When stations are relatively near each other so that broadcasting delay can be ignored, the assigned slot may immediately follow the corresponding minislot. Otherwise they are separated by other (mini)slots. In this scenario, if a station does not have anything to send, it will borrow the slot to another station. We shall first elaborate on this approach.

We shall first describe a fairly simple protocol for making a decision which station will transmit if the owner station does not do so. Let $p_1, p_2, \ldots, p_n$ be last reported numbers of packets to be sent, and let i be the station that just reported its packet number. There are two ways to interpret the number of reported packets to be transmitted: as an absolute number or as a relative number. In the first case, the reported number is the current number of messages in the queue for the transmission, while in the second case it is the number of newly arrived packets in the transmission queue. Protocol is similar in both cases, and we shall describe the protocol for the former, absolute case.

If $p_i \geq 1$ then station i sends one packet in the coming slot. Otherwise, let j be the first number among $i+1, i+2, ..., n, 1, 2, ..., i-1$ such that $p_j \geq 1$. Station j sends a packet in the coming slot, and all the stations decrement their local value of p_j. Since all stations have the same information, there will be no collision. The computation time needed to process the information can be neglected compared to communication time for transmitting the packet.

We shall now elaborate on the first approach, where stations do not directly owe any slots. In this approach, all the slots are equally available to all the participating stations under a given protocol. Let $p_1, p_2, ..., p_n$ be last reported numbers of packets to be sent, and let i be the station that just reported its packet number. Also, let j be the station that used the previous slot for transmission. The counter j follows a round-robin scheme. Thus, the next j is be the first among $j+1, j+2, ..., n, 1, 2, ..., j-1$ such that $p_j \geq 1$.

The later approach has an advantage, because it gives a better distribution of the available bandwidth. In the former approach, a station might be "lucky" if its left neighbor does not send much, whereas another station that has also significant demands for transmission can not gain anything from that spare bandwidth.

The second approach can be improved further. Although it gives the equal bandwidth to all the stations, it does not consider the transmission delay. Ideally, we want to keep track when each packet came to the transmission queue and the one that is the oldest would be transmitted first. To implement that, all stations simulate a kind of a centralized FIFO queue. One field in such a queue would contain reported numbers of packets to be sent from one of N stations within one periodic frame. The queue would be two-dimensional, where one entry (row) would consist of N fields containing reported numbers of all stations within one frame. This time however, those reported numbers will be relative numbers. Packets to be sent are always chosen from the oldest entry in the queue, to minimize the delay. Since we may have a number of packets in the same queue entry, round robin scheme would be used to give equal chance to all the stations that reported packets in that queue entry. With this approach, we have equal bandwidth assigned to all the station and equal distribution of delays (if, temporarily, the incoming traffic flow exceeds that the offered outgoing bandwidth).

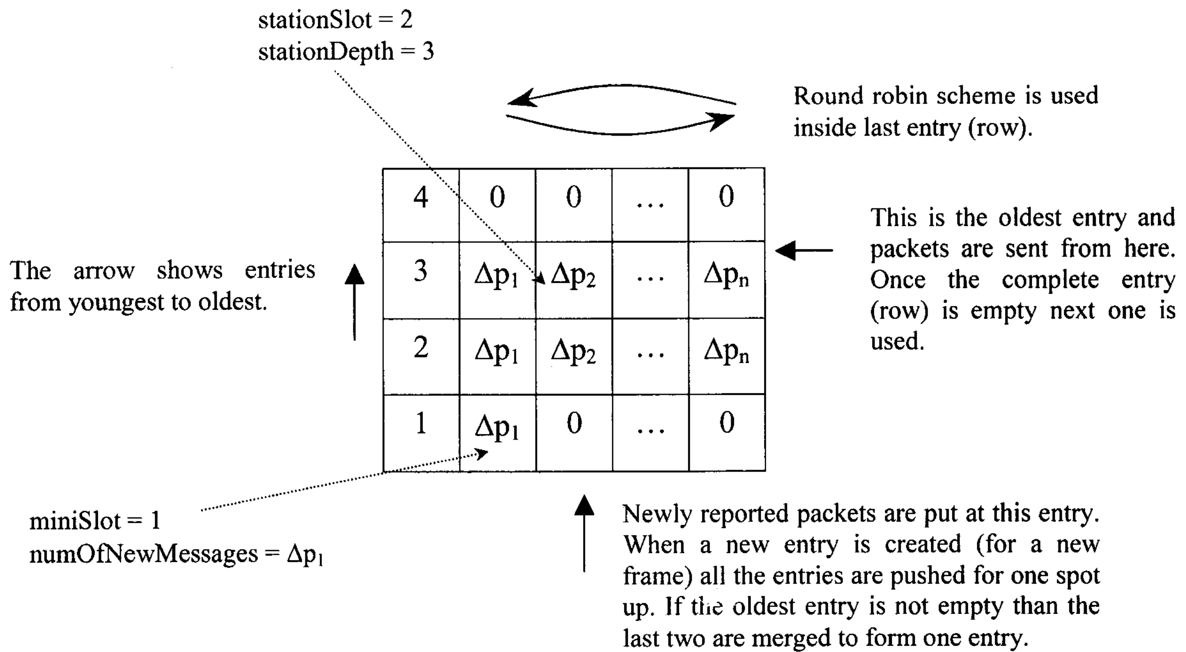

Fig. 8 Equal channel distribution

The corresponding protocol can be described as follows.

<u>Protocol Next-Station-To-Transmit</u>

```
/* definitions of variables */

Queue [MAX_NUM_OF_STATIONS] [MAX_DEPTH]; /* incoming queue of "advertised"
messages for each station in the last MAX_DEPTH periods */
miniSlot← 1; /* current minislot */
stationSlot ← 1; /* current slot (i.e. id of a station next to transmit a data packet) */
stationDepth ← MAX_DEPTH; /* current depth in the Queue */
N;    /* current number of stations */
numOfNewMessages;   /* number of newly arrived messages at local's station */

forever {
  get the new messages from the application level:
      store the messages into the station's data queue and the number
      of newly arrived ones store into numOfNewMessages;
   /* station with the id equals to the current minislot: broadcast the number of
     new incoming messages*/
  if (myId = miniSlot)
     TransmitToMiniSlotOfStation(miniSlot, numOfNewMessages);
```

```
  /* all the stations: read the current mini slot and store always at the bottom of Queue
*/
   Queue[miniSlot][1] ← ReceiveFromMiniSlotOfStation(miniSlot);

  /* find the next station to broadcast (a station from the "top" of the queue) */
  repeat
     /* try round robin inside the last row (depth) of the queue */
     repeat
        stationSlot ← stationSlot + 1;
        if (stationSlot = N + 1) then
           stationSlot ← 1;
        /* if the station has something really pending at that depth, it is indeed the next
one */
        if (Queue[stationSlot][stationDepth]  <> 0) {
           Queue[stationSlot][stationDepth]
             ← Queue[stationSlot][stationDepth] -1;
            break  one level of repeat loop;
        }
     until row at stationDepth is empty;
     /* inner loop was left either because we found the next station or the row was
empty */
     if  loop left because the row was empty
        stationDepth ← stationDepth – 1; /* try one level less */
      /* unless it is the bottom of the queue (level 1 row was empty) */
  until stationDepth = 0;

  /* if the queue was non-empty, we found the next station */
  if (stationDepth <> 0) then
       /* station with the id equals to the current slot: broadcast */
       if (myId = stationSlot) then
            broadcast next message from the data queue;

  /* update the "counters" and Queue */
  miniSlot ← miniSlot + 1;
  if (miniSlot > N) then {
     miniSlot ← 1;

     /* collapse the last two rows into one (really needed if Queue is filled up to that
level) */
     for (i ← 1; i <= N; i++)
         Queue[i][ MAX_DEPTH] ←
         Queue[i][MAX_DEPTH] + Queue[i][MAX_DEPTH-1];

     /* move up the rest of the rows by one */
     for (i ← 1; i <= N; i++)
          for (j ← 2; j <= MAX_DEPTH - 1; j++)
```

```
                Queue[i][j] ← Queue[i][j - 1];

        /* zero out the first row */
        for (i ← 1; i <= N; i++)
            Queue[i][1] ← 0;

        /* increment the current row where the next transmitting station is */
        stationDepth ← stationDepth + 1;

    } /* end of if */
} /* endforever */
```

Channel borrowing schemes have been already studied. The idea of minislots used for reservation purposes is well known and used in a number of the study cases (for example in [KS]). More sophisticated ideas with incoming queue were also considered. Rubin [R] assumed an existence of an incoming queue, but did not go into details of the algorithm how a next packet would be chosen, but was rather concentrated on different schemes of minislots. For example, he considered more complicated schemes of minislot assignment, where not all the stations would be granted a fixed minislot. We assumed the simplest case, were a station is assigned a fixed minislot, and focused on the detailed queue organization. Although not explicitly stated Rubin probabli assumed a singe FIFO queue. Our approach is multiply FIFO queues. Of course, this new approach will not decrease the average delay of packets. The idea is just to be fairer to all the stations. With the single-queue solution, if a station announce a large number of packets to transmit it will hold other stations, even they have just a few packets to transmit. Out approach gives a more fair distribution of the bandwidth to all stations (the approach is similar to a CPU round-robin scheduling with time-slicing for multitasking environment).

8. Conclusion

Multiple-access schemes have been topics of many research papers, where a different protocols were proposed on how to organize a group of station to send messages in TDMA fashion to broadcast medium. We gave a sort review of such papers, but all these papers assumed that the stations have their identity and echo knows what is the bandwidth (TDMA slot) assigned to them. Here we actually focused on how to initialize such a group of stations. The ultimate goal is that all the stations at the end of the initialization know how many users are in the group (if that was unknown) and what is exactly their position in the TDMA periodic frame. That means to get a number from *1* to *N*, where *N* is the number of stations in the group.

There were two parameters when we considered the initialization algorithm. This first one was whether the number of users is known or not. Although in a distributed environment it is more realistic to assume that the stations do not know the number of them in advance, the algorithms with known numbers of stations are also important. Actually, some of the algorithms with unknown number of stations are just modification of those with the known number of stations, where the number of stations is estimated using some techniques.

The second parameter is whether the initialization order is deterministic (any execution of the algorithm will give the same initialization order) for a given group of stations or it is randomized.

Such initialization algorithms were initially investigated in [HNO], and some of them were used for comparison and as a basis for further modification and improvement. They proposed an interesting algorithm that assumed randomized order and unknown number of users (*HNO-unknown-N*) and another one in the category of randomized with known number of users (*HNO-known-N*). Our first contribution is that we added a couple of enhancements that significantly improved the performance of the unknown-n algorithm (*HNO-unknown-N-mod*). We also proposed the unknown-n version of their known-n algorithm where the number n is actually estimated (*Estimated-N*). Our most important contribution is a new hybrid algorithm in both known-N and unknown-N flavors. It additionally improves performances of the above algorithms from approximately $e \cdot N$ to $2.2 \cdot N$.

Deterministic algorithms were not initially investigated and we proposed two of them. One is a deterministic version of *HNO-unknown-N-mod* (*HNO-det*), and another one completely new (*Three-det*). *HNO-det* offers insignificantly better performances and both are comparable with randomized version of the algorithms.

At the end we proposed some of our ideas how to utilize TDMA multiple access schemes in a broadcast medium, assuming a dynamic environment. We considered dynamic bandwidth requirements and dynamic number of station in a group.

For sure, there are still open issues and further enhancements in this work. Our focus was to increase performances of the algorithms not paying attention to their robustness. In the distributed environment it is very important that all the parties receive the same information and to be able to interpret them and proceed with the same action. If one party misses a broadcast message the things can easily go out of sync. That robustness can be improved introducing some kind of acknowledge messages. There is also completely another model of broadcast medium where noise and silence is not distinguished on the receiver side. The model is already considered in [HNO] for some types of algorithms. The algorithms are generally harder to come up with, but are more robust.

The algorithms were not analytically analyzed but we measured the performances in the simulations. One open problem is to theoretically find the bounds of the algorithms and find an exact formula for the complexity of our hybrid algorithms (we claim an approximate result of $2.2 \cdot N$).

9. References

[A] N. Abramson (ed.), Multiple access communications: Foundations for emerging texhnologies, IEEE Press, 1993.

[B] R. Binder, A dynamic packet switching system for satellite broadcast channels, in Proc. ICC'75, Vol. III, 41A, 1-5, San Francisco, CA, June 1975.

[CFL] I. Chlamtac, W.R. Franta and K.D. Levin, BRAM: The broadcast recognizing access method, IEEE Transactions on Communications, COM-27, 8, 1979, 1183-190.

[HNO] T. Hayashi, K. Nakano and S. Olariu, Randomized initialization protocols for packet radio networks, manuscript, 1998.

[IVB] T. Imielinski, S. Viswanathan and B.R. Badrinath, Data on air: organization and access, IEEE Trans. On Knowledge and data Engineering, 9, 3, 1997, 353-371.

[JM] S. Jangi, L.F. Merakos, Performance analysis of reservation random access protocols for wireless access networks, IEEE Trans. On Communications, 42, 2/3/4, 1994, 1223-1234.

[KM] O. Kubbar and H.T. Mouftah, Multiple access control protocols for wireless ATM: problems definition and design objectives, IEEE Communications Magazine, Nov. 1997, 93-99.

[KS] L. Kleinrock and M. Scholl, Packet switching in radio channels: New conflict-free multiple access schemes for a small number of data users, Proc. Int. Conf. On Communications, 1977, Vol. 2, 22.1., 105-11, 1977.

[K] O.S. Kosovych, Fixed Assignment access technique, IEEE Transactions on Communications, COM-26, 9, 1978, 1370-1376.

[NGS] S. Norskov, U. Gliese and K. Stubkjaer, Adaptive packet reservation multiple access (A-PRMA) for broadband wireless ATM, in: Mobile Multimedia Communications, Plenum Press, 1997, 167-171.

[OSZ] S. Olariu. I. Stojmenovic and A. Zomaya, On the dynamic initialization of parallel computers, Journal of Supercomputing, to appear.

[RS] R. Rom and M. Sidi, Multiple access protocols: Performance analysis, Springer-Verlag, 1985.

[R] I. Rubin, Access-control disciplines for multi-access communication channels: Reservation and TDMA schemes, IEEE Transactions on Information Theory, IT-25, 5, 1979, 516-536.

[S] F. C. Schoute, Decentralized control in packet switched satellite communication, IEEE Transactions on Automatic Control, vol. AC-23, No. 2, 1978, 362-371.

[T] F.A. Tobagi, Multiaccess protocols in packet communication systems, IEEE Transactions on Communications, COM-28, 4, 1980, 468-488.

[W] D.E. Willard, Log-logarithmic selection resolution protocols in a multiple access channel, SIAM J. on Computing, 15, 1986, 468-477.

Appendix A - Source code

Here we present full listings of the algorithms explained in the thesis:

- *HNO-known-N*
- *HNO-unknown-N-mod*
- *Estimated-N*
- *Hybrid-known-N*
- *Hybrid-estimated-N*
- *Three-det*

The code is written in C++. Except the generator of random number, which is UNIX specific, the code is generic and should run on any other platform.

```
//
// Protocol HNO-known-n
// ===================
//


#include <stdlib.h>  // used for rand()
#include <iostream.h>

#define numOfExperiments 100
#define N 100      // # of users

int counterDone = N;


//
//  class Ether
//

class Ether {
public:
  void messageBroadcasted(int channel, int type);
};


Ether theEther;


//
//  class User
//

class User {
public:
  User ();
  void clockUp();
  void clockDown();
  void listenMessage(int channel, int type);
  void print();
private:
  int m;
  int myId;
  int c;         // c is used in stage 1, by the end of stage 1 it gets the value of # of users in
my subgroup (all the users with the same channel)
  int myChannel, sendChannel, receiveChannel; // not really used
  int numOfMessages;  // number of messages attempted to be sent to me in one slot; if more that
one it was a collision
  typedef enum {
     startStage1,
     passiveStage1,
     end
  } State;
  State state;
  int sent;         // flag which says if a user sent a message in a given slot
};


User *pUser;



void Ether::messageBroadcasted(int channel, int type) {
  int i;

  for (i = 0; i < N; i++) {                  // ether gets a message from a user and forwards to
all the users
    pUser[i].listenMessage(channel, type);
  }
}
```

```
User::User() {          // constructor: initializes a user
  m = N; // known number of users
  c = 0;
  myId = 0;
  myChannel = sendChannel = receiveChannel = 1;
  numOfMessages = 0;
  state = startStage1;
  sent = 0;           // False
}


void User::clockUp() {

  switch (state) {

    case startStage1:

      if (lrand48() % m == 0) {  // active users send a message with probability of 1/m
        theEther.messageBroadcasted(sendChannel, 1);  // type 1
        sent = 1;     // True
      }
      break;

    case passiveStage1:
      break;

    case end:                  // all the users are assigned ID; a game starts

      receiveChannel = myChannel;  // restore...

      break;

  }  // end of switch

}  // end of User::clockUp()


void User::clockDown() {

  switch (state) {

    case startStage1:
    case passiveStage1:

      switch (numOfMessages) {

        case 0:           // silent

          break;

        case 1:           // unique

          m = m - 1;
          c = c + 1;
          if (sent) {    // if I am the one who sent the message
            myId = c;
            state = passiveStage1;
          }
          break;

        default:         // noise

          break;

      } // end of inner switch (numOfMessages)

      if (m < 1) {
```

```
        state = end;
        counterDone--;
      }

      break;

    case end:            // all the users are assigned ID; now, a game starts

      break;

  }  // end of outter switch(state)

  numOfMessages = 0;     // resets for the next slot
  sent = 0;

}  // end of User::clockDown()



void User::listenMessage(int channel, int type) {

  if (channel != receiveChannel)          // listen only broadcasted messages on my channel
    return;

  numOfMessages++;                    // even the sender will receive the message
} // end of User::listenMessage



void User::print() {
  cout << "sendChannel: " << sendChannel << "  m: " << m << " c: " << c << "  myId: " <<  myId <<
"  counterDone: " << counterDone << "  state: " << (int)state << "\n";
}



//
// main loop
//

main () {
  int i;
  int randomSeed;
  double avgStep = 0;
  int experiment;

  cin >> randomSeed;
  srand48(randomSeed);

  for (experiment = 0; experiment < numOfExperiments; experiment++) {

  pUser = new User[N];
  counterDone = N;
  int step = 0;

  while (1) {

    // cout << "-------------------\n";
    for (i = 0; i < N; i++)
      pUser[i].clockUp();          // slot is started with clockUp

    for (i = 0; i < N; i++)
      pUser[i].clockDown();        // slot is ended with clockDown

    for (i = 0; i < N; i++)       // print info between two slots
      ; // pUser[i].print();

    step++;

    if (counterDone == 0) break;
```

```
    } // end of while
  avgStep += step;
  cout << "\nsteps: " << step << "\n";
  } // end of experiments
  avgStep = avgStep / numOfExperiments;
  cout << "avgSteps: " << avgStep << "  N: " << N << "\n";
}
```

```
//
// Protocol HNO-unknown-n-mod
// ========================
//

#include <stdlib.h>  // used for rand()
#include <iostream.h>

#define numOfExperiments 100
#define N 1000      // # of users

int counterDone;

typedef int Data;  // type of data transmitted with a message; actually, should be a row of bits

//
//  class Ether
//

class Ether {
public:
  void messageBroadcasted(int channel, int data);
};

Ether theEther;

//
//  class User
//

class User {
public:
  User ();
  void clockUp();
  void clockDown();
  void listenMessage(int channel, Data data);
  void print();
private:
  int myId;
  Data dataReceived;
  int numOfMessagesReceived;
  int sendChannel, receiveChannel;
  typedef enum {
    partition1,
    partition0,
  } State;
  State state;
  int sent;
  int status1, status0;
  int l, L, NN, absPL;
  int active;
};

User *pUser;

void Ether::messageBroadcasted(int channel, Data data) {
  int i;

  for (i = 0; i < N; i++) {      // ether gets a message from a user and forwards to all the users
    pUser[i].listenMessage(channel, data);
  }
```

```
}


User::User() {          // constructor: initializes a user
  myId = 0;
  status1 = 2; // just to be > 1
  l = L = NN = 1;
  state = partition1;
  sendChannel = receiveChannel = 1;
  active = 1; // means uninitialized
}


void User::clockUp() {

  switch (state) {

    case partition1:
      if (!active) {
        sent = 0;
        break;
      }
      if (l < L) break;  // users from L - 1, L - 2,... are just listening, although still active
      sent = lrand48() % 2;
      if (sent)
        theEther.messageBroadcasted(1, 1);  // channel 1, data dummy
      break;

    case partition0:
      if (!active) break;
      if (l < L) break;
      if (!sent)
        theEther.messageBroadcasted(1, 1);  // channel 1, data dummy
      break;

  }  // end of switch

}  // end of User::clockUp()


void User::clockDown() {

  switch (state) {

    case partition1:
      status1 = numOfMessagesReceived;
      if (status1 == 1) {
        if (sent) {
          myId = NN;
          active = 0;
        }
        NN = NN + 1;
      }
      if (numOfMessagesReceived == 0)
        state = partition1;   // empty partition, try again
      else
        state = partition0;   // non-empty partition, check the other one
      break;

    case partition0:
      if (numOfMessagesReceived == 1) {
        if (active && l==L && !sent) {
          myId = NN;
          active = 0;
        }
        NN = NN + 1;
      }
      if (status1 < 2 && numOfMessagesReceived < 2)
```

```
        L = L - 1;
      if (status1 == 2 && numOfMessagesReceived == 2) {
        L = L + 1;
        if (sent) l = L;
      }
      state = partition1;

      break;

  }  // end of switch

  if (L == 0)
    counterDone--;
  numOfMessagesReceived = 0;
}  // end of User::clockDown()


void User::listenMessage(int channel, Data data) {

  if (channel != receiveChannel)    // listen only on one channel
    return;

  dataReceived = data;
  if (numOfMessagesReceived < 2) numOfMessagesReceived++;    // # == 2, there is a collision

} // end of User::listenMessage



void User::print() {
  cout << "myId: " <<  myId << "  l: " << l << "  L: " << L << "  NN: " << NN << "  s0: " <<
status0 << "  s1: " << status1 << "  sent: " << sent << "  state: " << (int)state << "
counterDone: " << counterDone << "\n";
}



//
// main loop
//

main () {
  int i;
  int randomSeed;
  double avgStep = 0;
  int experiment;

  cin >> randomSeed;
  srand48(randomSeed);

  for (experiment = 0; experiment < numOfExperiments; experiment++) {

  pUser = new User[N];
  counterDone = N;
  int step = 0;

  while (1) {

    // cout << "-------------------\n";
    for (i = 0; i < N; i++)
      pUser[i].clockUp();          // slot is started with clockUp

    for (i = 0; i < N; i++)
      pUser[i].clockDown();        // slot is ended with clockDown

    for (i = 0; i < N; i++)       // print info between two slots
      ; // pUser[i].print();

    step++;
```

```
    if (counterDone == 0) break;

    } // end of while

  avgStep += step;

  delete [] pUser;

  cout << "\nsteps: " << step << "\n";

  } // end of experiments

  avgStep = avgStep / numOfExperiments;

  cout << "avgSteps: " << avgStep << "  N: " << N << "\n";

}
```

```
//
// Protocol Estimated-n
// ====================
//

#include <stdlib.h>  // used for rand()
#include <iostream.h>

#define numOfExperiments 1
#define N 10     // # of users

int counterDone = N;

typedef int Data;  // type of data transmitted with a message; actually, should be a row of bits

//
//  class Ether
//

class Ether {
public:
  void messageBroadcasted(int channel, int type, int data);
};


Ether theEther;

//
//  class User
//

class User {
public:
  User ();
  void clockUp();
  void clockDown();
  void listenMessage(int channel, int type, Data data);
  void print();
private:
  int m, prevM;
  int myId;
  int c;   // c is used in stage 1, by the end of stage 1 it gets the value of # of users in my
subgroup (all the users with the same channel)
  int myChannel, sendChannel, receiveChannel;
  int numOfMessages;  // number of messages attempted to be sen to me in one slot; if more that
one it was a collision
  Data dataReceived;
  int typeReceived;
  typedef enum {
     startStage1,
     passiveStage1,
     end
  } State;
  State state;
  int sent;        // flag which says if a user sent a message in a given slot
  int step;
};

User *pUser;


void Ether::messageBroadcasted(int channel, int type, Data data) {
  int i;

  for (i = 0; i < N; i++) {                  // ether gets a message from a user and forwards to
all the users
    pUser[i].listenMessage(channel, type, data);
  }
}
```

```
User::User() {          // constructor: initializes a user
  m = 2; // m = 2; passive estimation
  c = 0;
  myId = 0;
  myChannel = sendChannel = receiveChannel = 1;
  numOfMessages = 0;
  state = startStage1;
  sent = 0;            // False
  step = 0;
  prevM = 0;
}


void User::clockUp() {

  switch (state) {

    case startStage1:

      if (lrand48() % m == 0) {  // active users send a message with probability of 1/m
        theEther.messageBroadcasted(sendChannel, 1, m);  // type 1, data m
        sent = 1;    // True
      }
      break;

    case passiveStage1:
      break;

    case end:                  // all the users are assigned ID; a game starts

      break;


  }  // end of switch

}  // end of User::clockUp()


void User::clockDown() {

  switch (state) {

    case startStage1:
    case passiveStage1:

      switch (numOfMessages) {

        case 0:            // silent

          if (step >= 0)
            step = -1;
          else
            step += step;
          m = m + step;
          break;

        case 1:            // unique

          m = m - 1;
          c = c + 1;
          step = 0;
          if (sent) {    // if I am the one who sent the message
            myId = c;
            state = passiveStage1;
          }
          break;

        default:          // noise

          if (step <= 0)
            step = 1;
```

```
        else
          step += step;
        m = m + step;


      } // end of inner switch (numOfMessages)

      if (m < 1)
        if (prevM == 1) {
          state = end;
          counterDone--;
        } else
          m = 1;
      prevM = m;

      break;

  }  // end of outter switch(state)

  numOfMessages = 0; // resets for the next slot
  sent = 0;
  dataReceived = 0;

}  // end of User::clockDown()


void User::listenMessage(int channel, int type, Data data) {

  if (channel != receiveChannel)           // listen only broadcasted messages on my channel
    return;

  numOfMessages++;                    // even the sender will receive the message
  dataReceived = data;
  typeReceived = type;
} // end of User::listenMessage



void User::print() {
  cout << "sendChannel: " << sendChannel << "  m: " << m << " c: " << c << "  myId: " <<  myId <<
"  counterDone: " << counterDone << "  state: " << (int)state << "\n";
}



//
// main loop
//

main () {
  int i;
  int randomSeed;
  double avgStep = 0;
  int experiment;

  cin >> randomSeed;
  srand48(randomSeed);


  for (experiment = 0; experiment < numOfExperiments; experiment++) {

  pUser = new User[N];
  counterDone = N;
  int step = 0;

  while (1) {

    cout << "-------------------\n";
    for (i = 0; i < N; i++)
      pUser[i].clockUp();         // slot is started with clockUp
```

```
    for (i = 0; i < N; i++)
      pUser[i].clockDown();        // slot is ended with clockDown

    for (i = 0; i < N; i++)       // print info between two slots
      pUser[i].print();

    step++;

    if (counterDone == 0) break;

  } // end of while

  avgStep += step;

  delete [] pUser;

  cout << "\nsteps: " << step << "\n";

  } // end of experiments

  avgStep = avgStep / numOfExperiments;

  cout << "avgSteps: " << avgStep << "  N: " << N << "\n";

}
```

```
//
// Protocol Hybrid-known-n
// =====================
//

#include <stdlib.h>  // used for rand()
#include <iostream.h>


#define numOfExperiments 100
#define N 30     // # of users


int flagDone[N];
int counterDone = N;


//
//  class Ether
//

class Ether {
public:
  void messageBroadcasted(int channel, int type);
};



Ether theEther;



//
//  class User
//

class User {
public:
  User ();
  void clockUp();
  void clockDown();
  void listenMessage(int channel, int type);
  void print();
private:
  int m, prevM;
  int myId;
  int c;        // c is used in stage 1, by the end of stage 1 it gets the value of # of users in
my subgroup (all the users with the same channel)
  int myChannel, sendChannel, receiveChannel;
  int numOfMessagesReceived;  // number of messages attempted to be sent to me in one slot; if
more that one it was a collision
  typedef enum {
     startStage,
     passiveStage1,
     partition1,
     partition0,
     end
  } State;
  State state;
  typedef enum {
     activeStage,
     passiveStage2,
     passiveStage3
  } SubState;
  SubState subState;
  int sent;        // flag which says if a user sent a message in a given slot
  int l, L, NN;
  int status0[100], is, status1;
};
```

```
User *pUser;


void Ether::messageBroadcasted(int channel, int type) {
  int i;

  for (i = 0; i < N; i++) {                    // ether gets a message from a user and forwards to
all the users
    pUser[i].listenMessage(channel, type);
  }
}


User::User() {          // constructor: initializes a user
  m = N; // known number of users
  c = 0;
  myId = 0;
  myChannel = sendChannel = receiveChannel = 1;
  numOfMessagesReceived = 0;
  state = startStage;
  sent = 0;           // False
  prevM = 0;
}


void User::clockUp() {

  switch (state) {

    case startStage:

      if (rand() % m < 15) {  // active users send a message with probability of 1/m
        theEther.messageBroadcasted(sendChannel, 1);  // type 1
        sent = 1;     // True
      } else
        sent = 0;
      break;

    case passiveStage1:
        sent = 0;
      break;

    case partition1:
      if (subState != activeStage) break;
      if (l < L) break;  // users from L - 1, L - 2,... are just listening
      sent = rand() % 2;
      // cout << "1:  sent: " << sent << "\n";
      if (sent)
        theEther.messageBroadcasted(1, 1);  // channel 1, data dummy
      break;

    case partition0:
      if (subState !=activeStage) break;
      if (l < L) break;
      if (!sent)
        theEther.messageBroadcasted(1, 1);  // channel 1, data dummy
      break;

    case end:                  // all the users are assigned ID; a game starts

      receiveChannel = myChannel;  // restore...

      break;


  }  // end of switch
```

```
}  // end of User::clockUp()


void User::clockDown() {

  switch (state) {

    case startStage:
    case passiveStage1:

      switch (numOfMessagesReceived) {

        case 0:           // silent

//          m = m - 1;
          break;

        case 1:           // unique

          m = m - 1;
          c = c + 1;
          if (sent) {    // if I am the one who sent the message
            myId = myId + c;
            state = passiveStage1; // end just continue listening
          }
          break;

        default:         // noise

//          m = m + 1;
  // collided ones continue resolution of IDs; the others listen and wait them to finish
    if (sent == 1)
      subState = activeStage;
    else if (state == startStage)
      subState = passiveStage2;
    else /* state == passiveStage1 */
      subState = passiveStage3;
    is = 0;
    status1 = status0[is] = 2; // just to be > 1
    l = L = 1;
    NN = c + 1;
    state = partition1;
          break;


      } // end of inner switch (numOfMessagesReceived)

      // cout << "2:  m: " << m << "  prevM: " << prevM << "\n";
      if (m < 1)
        if (1 /* prevM == 1 */) { //obsolite for known N
          state = end;
          counterDone--;
          // cout << "3:    sendChannel: " << sendChannel << "  myId: " << myId << "\n";
        } else
          m = 1;
      prevM = m;

      break;

    case partition1:
      status1 = numOfMessagesReceived;
      if ((numOfMessagesReceived) == 0)
        state = partition1;   // empty partition, try again
      else
        state = partition0;   // non-empty partition, check the other one
      break;

    case partition0:
      if ((numOfMessagesReceived) == 0)
        state = partition1;   // empty partition, try again
```

```
      else {
        status0[++is] = numOfMessagesReceived;
        L = L + 1;
        if (sent) l = L;       // marking at which level we are active
        state = partition1;   // non-empty partition
        while (status1 == 1) {
          if (l == L && subState == activeStage) {      // not those that listen, only active
ones get assigned ID
            myId = NN;
            // state = end;
            subState = passiveStage3;
            // counterDone--;
            // break;          realy necessary?
          }
          NN = NN + 1;      // others that listen, just update parameters
          L = L - 1;
          status1 = status0[is--];
        }
        if (is < 0) {
          if (subState == passiveStage2)
            state = startStage;
          else /* subState == passiveStage2 */
            state = passiveStage1;
          c = NN - 1;
          m = N - c;
        }
      }
      break;

    case end:                  // all the users are assigned ID; now, a game starts

      break;

  }  // end of outter switch(state)

  numOfMessagesReceived = 0; // resets for the next slot
//   sent = 0;

}  // end of User::clockDown()



void User::listenMessage(int channel, int type) {

  if (channel != receiveChannel)           // listen only broadcasted messages on my channel
    return;

  numOfMessagesReceived++;                    // even the sender will receive the message
} // end of User::listenMessage



void User::print() {
  cout // << "sendChannel: " << sendChannel
       << "  m: " << m
       << "  NN: " << NN
       << "  c: " << c
       << "  sent: " << sent
       << "  myId: " <<  myId
       << "  counterDone: " << counterDone
       << "  state: " << (int)state
       << "  subState: " << (int)subState
       << "  l: " << l
       << "  L: " << L
       << "  is: " << is
       << "\n";
}



//
```

```
// main loop
//
main () {
  int i;
  int randomSeed;
  double avgStep = 0;
  int experiment;

  cin >> randomSeed;
  srand48(randomSeed);

  for (experiment = 0; experiment < numOfExperiments; experiment++) {

  pUser = new User[N];
  counterDone = N;
  int step = 0;

  while (1) {

    // cout << "-------------------\n";
    for (i = 0; i < N; i++)
      pUser[i].clockUp();         // slot is started with clockUp

    for (i = 0; i < N; i++)
      pUser[i].clockDown();       // slot is ended with clockDown

    for (i = 0; i < N; i++)      // print info between two slots
      ; // pUser[i].print();

    step++;

    if (counterDone == 0) break;

    } // end of while

  avgStep += step;

  cout << "\nsteps: " << step << "\n";

  } // end of experiments

  avgStep = avgStep / numOfExperiments;

  cout << "avgSteps: " << avgStep << "  N: " << N << "\n";

}
```

```
//
// Protocol Hybrid-estimated-n
// ==========================
//

#include <stdlib.h>  // used for rand48()
#include <iostream.h>


#define numOfExperiments 1
#define N 5000    // # of users


int flagDone[N];
int counterDone;


//
//  class Ether
//

class Ether {
public:
  void messageBroadcasted(int channel, int type);
};



Ether theEther;



//
//  class User
//

class User {
public:
  User ();
  void clockUp();
  void clockDown();
  void listenMessage(int channel, int type);
  void print();
private:
  int m, prevM;
  int myId;
  int myChannel, sendChannel, receiveChannel;
  int numOfMessagesReceived;  // number of messages attempted to be sent to me in one slot; if
more that one it was a collision
  typedef enum {
     estimation,
     activeState,
     passiveState,
     partition1,
     partition0,
     end
  } State;
  State state;
  typedef enum {
     activeSubState,
     passiveSubState1,
     passiveSubState2
  } SubState;
  SubState subState;
  int sent;        // flag which says if a user sent a message in a given slot
  int sentInP1, sentInP0;
  int l, L, NextId;
  int status0, status1;
  int step;
};
```

```
User *pUser;


void Ether::messageBroadcasted(int channel, int type) {
  int i;

  for (i = 0; i < N; i++) {    // ether gets a message from a user and forwards to all the users
    pUser[i].listenMessage(channel, type);
  }
}


User::User() {          // constructor: initializes a user
  m = 2; // uknown number of users
  NextId = 1;
  myId = 0;
  myChannel = sendChannel = receiveChannel = 1;
  numOfMessagesReceived = 0;
  state = estimation;
  sent = 0;           // False
  prevM = 0;
  step = 0;
}


void User::clockUp() {

  switch (state) {

    case estimation:
    case activeState:

      if (lrand48() % m < 1) {  // active users send a message with probability of 1/m
        theEther.messageBroadcasted(sendChannel, 1);  // type 1
        sent = 1;    // True
      } else
        sent = 0;
      break;

    case passiveState:
        sent = 0;
      break;

    case partition1:
      if (subState != activeSubState)
        break;
      if (l < L)
        break;  // users from L - 1, L - 2,... are just listening
      sentInP1 = lrand48() % 2;
      // cout << "1:  sent: " << sent << "\n";
      if (sentInP1)
        theEther.messageBroadcasted(1, 1);  // channel 1, data dummy
      break;

    case partition0:
      if (subState !=activeSubState) break;
      if (l < L) break;
      sentInP0 = !sentInP1;
      if (sentInP0)
        theEther.messageBroadcasted(1, 1);  // channel 1, data dummy
      break;

    case end:                  // all the users are assigned ID; a game starts

      receiveChannel = myChannel;  // restore...
```

```
      break;


  }  // end of switch

}  // end of User::clockUp()


void User::clockDown() {

  switch (state) {

    case estimation:
      switch (numOfMessagesReceived) {

        case 0:             // silent
          state = activeState;
          break;

        case 1:
          m = m - 1;
          if (sent) {     // if I am the one who sent the message
            myId = NextId;
            state = passiveState; // end just continue listening
          } else
            state = activeState;
          NextId = NextId + 1;
          break;

        default:
          m += m;
          break;
      } // end of switch
      break;

    case activeState:
    case passiveState:

      switch (numOfMessagesReceived) {

        case 0:             // silent

          if (step >= 0)
            step = -1;
          else
            step += step;
          m = m + step;
          break;

        case 1:             // unique

          m = m - 1;
          step = 0;
          if (sent) {     // if I am the one who sent the message
            myId = NextId;
            state = passiveState; // end just continue listening
          }
          NextId = NextId + 1;
          break;

        default:          // noise

          // collided stationss continue resolution of IDs; the others listen and wait them to
finish
          if (sent == 1)
            subState = activeSubState;
          else if (state == activeState)
            subState = passiveSubState1;
          else /* state == passiveState */
```

```
        subState = passiveSubState2;
      l = L = 1;

      if (step <= 0)
        step = 1;
      else
        step += step;
      m = m + step;

      state = partition1;

      break;


  } // end of inner switch (numOfMessagesReceived)

  // cout << "2:  m: " << m << "  prevM: " << prevM << "\n";
  if (m < 1)
    if (prevM == 1) {
      state = end;
      counterDone--;
      // cout << "3:    sendChannel: " << sendChannel << "  myId: " << myId << "\n";
    } else
      m = 1;
  prevM = m;

  break;

case partition1:
  status1 = numOfMessagesReceived;
  if (status1 == 1) {
    if (subState == activeSubState && sentInP1) {
      myId = NextId;
      subState = passiveSubState2;
    }
    NextId = NextId + 1;
    m = m - 1;
  }
  if (status1 == 0)
    state = partition1;   // empty partition, try again
  else    // status1 == 1 or 2
    state = partition0;   // non-empty partition, check the other one
  break;

case partition0:
  status0 = numOfMessagesReceived;
  if (status0 == 1) {
    if (subState == activeSubState &&  l==L &&  sentInP0) {
      myId = NextId;
      subState = passiveSubState2;
    }
    NextId = NextId + 1;
    m = m - 1;
  }
  if (status1 < 2 && status0 < 2)
    L = L - 1;
  if (status1 == 2 && status0 == 2) {
    L = L + 1;
    if (!sentInP0) l = L;
  }
  state = partition1;
  if (L == 0) {
      if (subState == passiveSubState1)
        state = activeState;
      else /* subState == passiveSubState2 */
        state = passiveState;
      if (m < 1) m = 1;
  }

  break;
```

```
    case end:                   // all the users are assigned ID; now, a game starts

      break;

  }  // end of outter switch(state)

  numOfMessagesReceived = 0; // resets for the next slot

}  // end of User::clockDown()


void User::listenMessage(int channel, int type) {

  if (channel != receiveChannel)          // listen only broadcasted messages on my channel
    return;

  if (numOfMessagesReceived < 2)          // # == 2, there is a collision
    numOfMessagesReceived++;              // even the sender will receive the message
} // end of User::listenMessage


void User::print() {
  cout // << "sendChannel: " << sendChannel
       << "  m: " << m
       << "  NextId: " << NextId
       << "  sent: " << sent
       << "  myId: " <<  myId
       << "  counterDone: " << counterDone
       << "  state: " << (int)state
       << "  subState: " << (int)subState
       << "  l: " << l
       << "  L: " << L
       << "\n";
}


//
// main loop
//
main () {
  int i;
  int randomSeed;
  double avgStep = 0;
  int experiment;

  cin >> randomSeed;
  srand48(randomSeed);

  for (experiment = 0; experiment < numOfExperiments; experiment++) {

  pUser = new User[N];
  counterDone = N;
  int step = 0;

  while (1) {

    // cout << "-------------------\n";
    for (i = 0; i < N; i++)
      pUser[i].clockUp();          // slot is started with clockUp

    for (i = 0; i < N; i++)
      pUser[i].clockDown();        // slot is ended with clockDown

    for (i = 0; i < N; i++)       // print info between two slots
      ;// pUser[i].print();

    step++;
```

```
    if (counterDone == 0) break;
    } // end of while
  avgStep += step;
  cout << "\nsteps: " << step << "\n";
  } // end of experiments
  avgStep = avgStep / numOfExperiments;
  cout << "avgSteps: " << avgStep << "  N: " << N << "\n";
}
```

```
//
// Protocol HNO-det
// ================
//

#include <stdlib.h>  // used for rand()
#include <iostream.h>

#define SER_NUM  (1 << (lengthOfSerialNum))   // # max of serial numbers
#define MAX_SER_NUM 5000    // max # of serial numbers
#define MAX_SER_LEN 64        // max length of a serial number

#define numOfExperiments 10


int lengthOfSerialNum  = 8;
int N = 32;      // # of users


int counterDone;

typedef int Data;  // type of data transmitted with a message; actually, should be a row of bits


//
// SerialNum
//

class SerialNum {
public:
  SerialNum();
  int getBit(int B);
  char *print(char *str);
  bool operator==(SerialNum &);

private:
  char bit[MAX_SER_LEN];
};

SerialNum::SerialNum() {
  for (int i = 0; i < lengthOfSerialNum; i++)
    bit[i] = lrand48() % 2;
}

int SerialNum::getBit(int B) {
  return (int)bit[lengthOfSerialNum - B];
}

bool SerialNum::operator==(SerialNum &serialNum) {
  for (int i = 0; i < lengthOfSerialNum; i++)
    if (bit[i] != serialNum.bit[i])
      return false;
  return true;
}

char *SerialNum::print(char *str) {
  int j = 0;

  for (int i = lengthOfSerialNum - 1; i >=0; i--, j++)
    str[j] = '0' + bit[i];

  str[j] = '\0';

  return str;
}

//
// SerialNumServer
//

class SerialNumServer {
```

```
public:
  SerialNumServer();
  SerialNum getUniqueSerialNum();
private:
  SerialNum used[MAX_SER_NUM];
  int numOfUsed;
};


SerialNumServer *pSerialNumServer;


SerialNumServer::SerialNumServer() {
  numOfUsed = 0;
}


SerialNum SerialNumServer::getUniqueSerialNum() {
  SerialNum serialNum;
  int i;

  do {
    for (i = 0; i < numOfUsed; i++)
      if (used[i] == serialNum)
        break;
    if (i == numOfUsed) break;
    serialNum = SerialNum();
  } while (1);
  return used[numOfUsed++] = serialNum;
}


//
//  class Ether
//

class Ether {
public:
  void messageBroadcasted(int channel, int data);
};



Ether theEther;



//
//  class User
//

class User {
public:
  User ();
  void clockUp();
  void clockDown();
  void listenMessage(int channel, Data data);
  void print();
private:
  SerialNum serialNum;
  int myId;
  Data dataReceived;
  int numOfMessagesReceived;
  int sendChannel, receiveChannel;
  typedef enum {
    partition1,
    partition0,
  } State;
  State state;
  int sent, sentIndex;
  int status1, status0;
```

```
  int l, L, NN, absPL;
  int active;
};


User *pUser;


void Ether::messageBroadcasted(int channel, Data data) {
  int i;

  for (i = 0; i < N; i++) {                    // ether gets a message from a user and forwards to
all the users
    pUser[i].listenMessage(channel, data);
  }
}



User::User() {         // constructor: initializes a user
  serialNum = pSerialNumServer->getUniqueSerialNum();
  myId = 0;
  status1 = 2; // just to be > 1
  l = L = NN = 1;
  state = partition1;
  sentIndex = 0;
  sendChannel = receiveChannel = 1;
  active = 1; // means uninitialized
}



void User::clockUp() {

  switch (state) {

    case partition1:
      if (!active) break;
      if (l < L) break;  // users from L - 1, L - 2,... are just listening, although still active
      sentIndex++;
      sent = serialNum.getBit(sentIndex);
      if (sent)
        theEther.messageBroadcasted(1, 1);  // channel 1, data dummy
      break;

    case partition0:
      if (!active) break;
      if (l < L) break;
      if (!sent)
        theEther.messageBroadcasted(1, 1);  // channel 1, data dummy
      break;

  }  // end of switch

}  // end of User::clockUp()



void User::clockDown() {

  switch (state) {

    case partition1:
      status1 = numOfMessagesReceived;
      if (status1 == 1) {
        if (sent) {
          myId = NN;
          active = sent = 0;
        }
        NN = NN + 1;
      }
```

```
      if (numOfMessagesReceived == 0)
        state = partition1;   // empty partition, try again
      else
        state = partition0;   // non-empty partition, check the other one
      break;

    case partition0:
      if (numOfMessagesReceived == 1) {
        if (active && l==L && !sent) {
          myId = NN;
          active = 0;
        }
        NN = NN + 1;
      }
      if (status1 < 2 && numOfMessagesReceived < 2)
        L = L - 1;
      if (status1 == 2 && numOfMessagesReceived == 2) {
        L = L + 1;
        if (sent) l = L;
      }
      state = partition1;

      break;

  }  // end of switch

  if (L == 0)
    counterDone--;
  numOfMessagesReceived = 0;
}  // end of User::clockDown()


void User::listenMessage(int channel, Data data) {

  if (channel != receiveChannel)   // listen only on one channel
    return;

  dataReceived = data;
  if (numOfMessagesReceived < 2) numOfMessagesReceived++;   // # == 2, there is a collision

} // end of User::listenMessage


void User::print() {
  cout << "myId: " << myId << "  l: " << l << "  L: " << L << "  NN: " << NN << "  s0: " <<
status0 << "  s1: " << status1 << "  sent: " << sent << "  state: " << (int)state << "
counterDone: " << counterDone << "\n";
}



//
// main loop
//

main () {
  int i;
  int randomSeed;
  double avgStep = 0;
  int experiment;

  cin >> randomSeed;
  srand48(randomSeed);

  for (experiment = 0; experiment < numOfExperiments; experiment++) {

  pSerialNumServer = new SerialNumServer;
  pUser = new User[N];
  counterDone = N;
```

```
  int step = 0;

  while (1) {

    // cout << "--------------------\n";
    for (i = 0; i < N; i++)
      pUser[i].clockUp();         // slot is started with clockUp

    for (i = 0; i < N; i++)
      pUser[i].clockDown();       // slot is ended with clockDown

    for (i = 0; i < N; i++)      // print info between two slots
      ; // pUser[i].print();

    step++;

    if (counterDone == 0) break;

    } // end of while

  avgStep += step;

  delete pSerialNumServer;
  delete [] pUser;

  cout << "\nsteps: " << step << "\n";

  } // end of experiments

  avgStep = avgStep / numOfExperiments;

  cout << "avgSteps: " << avgStep << "  N: " << N << "\n";

}
```

```
//
// Protocol Three-det
// ==================
//


#include <stdlib.h>  // used for rand()
#include <iostream.h>

#define numOfExperiments 10

#define SER_NUM  (1 << (lengthOfSerialNum))   // actual number of serial numbers
#define MAX_SER_NUM 5000     // max # of serial numbers
#define MAX_SER_LEN 64         // max length of a serial number

int lengthOfSerialNum  = 8;
int N = 32;  // # of users

#define _HARD_CODED_ 0

int counterDone;


typedef int Data;  // type of data transmitted with a message; actually, should be a row of bits


//
//  class Ether
//

class Ether {
public:
  void messageBroadcasted(int channel, int data);
};



Ether theEther;

class SerialNum {
public:
  SerialNum();
  int getBit(int B);
  char *print(char *str);
  bool operator==(SerialNum &);
  bool operator<(SerialNum &);
private:
  friend class SerialNumServer;
  char bit[MAX_SER_LEN];
};

SerialNum::SerialNum() {
  for (int i = 0; i < lengthOfSerialNum; i++)
    bit[i] = lrand48() % 2;
}

int SerialNum::getBit(int B) {
  return (int)bit[lengthOfSerialNum - B];
}

bool SerialNum::operator==(SerialNum &serialNum) {
  for (int i = lengthOfSerialNum - 1; i>=0; i--)
    if (bit[i] != serialNum.bit[i])
          return false;
  return true;
}

bool SerialNum::operator<(SerialNum &serialNum) {
  for (int i = lengthOfSerialNum - 1; i>=0; i--)
    if (bit[i] < serialNum.bit[i])
      return true;
```

```
        else if (bit[i] > serialNum.bit[i])
          return false;
  return false;
}

char *SerialNum::print(char *str) {
  int j = 0;

  for (int i = lengthOfSerialNum - 1; i >=0; i--, j++)
    str[j] = '0' + bit[i];

  str[j] = '\0';

  return str;
}

//
// SerialNumServer
//

class SerialNumServer {
public:
  SerialNumServer();
  SerialNum getUniqueSerialNum();
private:
  SerialNum used[MAX_SER_NUM];
  int numOfUsed;
};


SerialNumServer *pSerialNumServer;

int indexHardCodedSerials;

SerialNumServer::SerialNumServer() {
  numOfUsed = 0;
}


#if !_HARD_CODED_

SerialNum SerialNumServer::getUniqueSerialNum() {
  SerialNum serialNum;
  int i;

  do {
    for (i = 0; i < numOfUsed; i++)
      if (used[i] == serialNum)
        break;
    if (i == numOfUsed) break;
    serialNum = SerialNum();
  } while (1);
  return used[numOfUsed++] = serialNum;
}

#else

SerialNum SerialNumServer::getUniqueSerialNum() {
  SerialNum serialNum;
  int i;
  __int64 serial = HardCoddedSerials[indexHardCodedSerials];

  for (i = 0; i < lengthOfSerialNum; i++) {
          serialNum.bit[i] = (char)(serial & 0xf);
          serial = serial >> 4;
  }

  indexHardCodedSerials++;

  return serialNum;
}
```

```
#endif

//
//  class User
//

class User {
public:
  User ();
  void clockUp();
  void clockDown();
  void listenMessage(int channel, Data data);
  void print();
  void deleteIntermediateStats();
  int myId;
private:
  SerialNum serialNum;
  Data dataReceived;
  int numOfMessagesReceived;
  int sendChannel, receiveChannel;
  int bit, nextId, L, active, defeatLevel, sign;
  int stackNextBackward[100], spNextBackward, stackNextForward[100], spNextForward;
};



User *pUser;



void Ether::messageBroadcasted(int channel, Data data) {
  int i;

  for (i = 0; i < N; i++) {                  // ether gets a message from a user and forwards to
all the users
    pUser[i].listenMessage(channel, data);
  }
}

User::User() {         // constructor: initializes a user
  serialNum = pSerialNumServer->getUniqueSerialNum();
  nextId = 0; L = 1; myId = 0;
  active = 1;     // True
  receiveChannel = 1;
  numOfMessagesReceived = 0;
  dataReceived = 0;
  defeatLevel = 0;
  sign = 1;
  spNextForward = -1;
  spNextBackward = -1;
}


void User::deleteIntermediateStats() {
  numOfMessagesReceived = 0;
}

void User::clockUp() {

      if (!active)
        return;

      bit = serialNum.getBit(L);

      if (bit)
        theEther.messageBroadcasted(1, 0); // channel 1, content not important (0)

}  // end of User::clockUp()

void User::clockDown() {
```

```
switch (numOfMessagesReceived) {

  case 0:
    if (sign > 0)
      L = L + 1;
    else {
      if (spNextBackward >= 0)
        L = stackNextBackward[spNextBackward--];
      else {
        sign = 1;
        L = L + 1;
      }
     if (spNextForward >= 0)
        spNextForward--;
    }
    break;

  case 1:
    nextId = nextId + 1;
    if (bit) {
      myId = nextId;
      active = 0;
      defeatLevel = -1;
      bit = 0;
      counterDone--;
    }
      if (sign == 1)
        stackNextForward[++spNextForward] = L;
      else
        spNextForward--;
    if (spNextBackward >= 0) {
      L = stackNextBackward[spNextBackward--];
      sign = -1;
   } else {
      L = L + 1;
      sign = 1;
    }
    break;

  default:
    if (active && !bit) {
      active = 0;
      defeatLevel = L;
    }

    stackNextBackward[++spNextBackward] = L;

    if (sign == -1) {
      if (spNextForward >= 0) {
        L = stackNextForward[spNextForward--] + 1;
      } else
        L = L + 1;
    } else {
      stackNextForward[++spNextForward] = L;
      L = L + 1;
    }

    sign = 1;
    break;
}
if (defeatLevel == L) {
  defeatLevel = 0;
  active = 1;
}
if (L > lengthOfSerialNum) {
  if (myId == 0) {
    counterDone--;
    myId = ++nextId;
  }
  counterDone--;
```

```
      }

          dataReceived = 0;
}  // end of User::clockDown()


void User::listenMessage(int channel, Data data) {

  if (channel != receiveChannel)   // listen only on one channel
    return;

  dataReceived = data;
  numOfMessagesReceived++;         // if # > 1, there is a collision

} // end of User::listenMessage


void User::print() {
    char str[100];

    cout << "serNum: " << serialNum.print(str) << "  L: " << L << "  myId: " << myId
        << "  msgRcv: " << numOfMessagesReceived << "  nextId: " << nextId
        << "  act: " << active << "  bit: " << bit <<  "  dL: " << defeatLevel
        << "  sign: " << sign << "\n";
}



//
// main loop
//

void main () {
  int i, j;
  int randomSeed;
  double avgStep = 0;
  int experiment;


  cin >> randomSeed;
  srand48(randomSeed);

  for (experiment = 0; experiment < numOfExperiments; experiment++) {

  pSerialNumServer = new SerialNumServer;
  pUser = new User[N];
  counterDone = N;
  int step = 0;
  indexHardCodedSerials = 0;

  while (1) {

    // cout << "-------------------\n\n";

    for (i = 0; i < N; i++)
      pUser[i].deleteIntermediateStats();

    for (i = 0; i < N; i++)
      pUser[i].clockUp();         // slot is started with clockUp

    for (i = 0; i < N; i++)
      pUser[i].clockDown();       // slot is ended with clockDown

    for (i = 0; i < N; i++)
      ; // pUser[i].print();

    step++;
    // cout << "step: " << step << "  counterDone: " << counterDone << "\n";

    if (counterDone < 0) break;
```

```
  } // end of while

  // check if everything is OK
  for (i = 1; i <= N; i++) {
        for (j = 0; j < N; j++)
          if (pUser[j].myId == i) break;
        if (j == N) cout << "final check: " << i << " not assigned to anybody\n";
  }

  delete pSerialNumServer;
  delete pUser;

  avgStep += step;

  cout << "\nsteps: " << step << "\n";

  } // end of experiments

  avgStep = avgStep / numOfExperiments;

  cout << "avgSteps: " << avgStep << "  N: " << N << "  lenSerNum: " << lengthOfSerialNum <<
"\n";
}
```

Appendix B - Detailed simulation results

Deterministic

# of stations	3	10	30	100	300
ser. # length			*Tree-det*		
2	3.18				
4	6.04	16.18			
8	10.59	24.97	69.51	211.97	
16	19	33.34	83.25	268.02	796.07
32	35.57	49.56	101.32	290.08	811.99
64	66.78	83.35	136.12	328.6	844.56
			HNO-det		
2	4				
4	5.15	18.77			
8	6.13	23.49	70.83	217.04	
16	6.03	23.75	76.92	264.77	795.82
32	7.5	23.82	78.88	270.93	795.57
64	5.73	25.49	81.18	276.83	795.32

Randomized with known N

N	3	10	30	100	300	1000		
HNO-known-n								
average	5.1	22.695	75.626	263.979	806.062	2708.95		
deviation	2.1332	5.54894	10.682	21.0031	35.6166	71.1986		
average relative	1.7	2.2695	2.52087	2.63979	2.68687	2.70895		
lower bound - 95%	1.6559	2.23511	2.4988	2.62677	2.67951	2.70454		
upper bound - 95%	1.7441	2.30389	2.54294	2.65281	2.69423	2.71336		
lower bound - 99%	1.6421	2.2243	2.49186	2.62268	2.6772	2.70315		
upper bound - 99%	1.7579	2.3147	2.54987	2.6569	2.69654	2.71475		
Hybrid-known-n-mod								
average	5.002	19.842	62.85	215.998	652.732	2181.43		
deviation	2.006	4.44959	7.62467	15.2861	25.9279	47.1495		
average relative	1.6673	1.9842	2.095	2.15998	2.17577	2.18143		
lower bound - 95%	1.6259	1.95662	2.07925	2.15051	2.17042	2.17851		
upper bound - 95%	1.7088	2.01178	2.11075	2.16945	2.18113	2.18435		
lower bound - 99%	1.6129	1.94795	2.0743	2.14753	2.16873	2.17759		
upper bound - 99%	1.7218	2.02045	2.1157	2.17243	2.18281	2.18527		
Hybrid-known-n-mod-reduced (with the latest optimizations)								
average	4.834	19.32	61.675	212.905	641.945	2147.97		
deviation	1.7764	3.76028	6.92975	14.0032	23.1538	42.328		
average relative	1.6113	1.932	2.05583	2.12905	2.13982	2.14797		
lower bound - 95%	1.5746	1.90869	2.04152	2.12037	2.13503	2.14535		
upper bound - 95%	1.648	1.95531	2.07015	2.13773	2.1446	2.1506		
lower bound - 99%	1.5631	1.90137	2.03702	2.11764	2.13353	2.14453		
upper bound - 99%	1.6596	1.96263	2.07465	2.14046	2.1461	2.15142		
	0.8	1	1.2	1.5	2	4	8	15
Hybrid-known-n-mod-reduced (N = 30)								
	2.1214	2.05583	2.06887	2.11357	2.25183	2.6613	2.90927	3.01227

Randomized with unknown N

N	3	10	30	100	300	1000
Estimated-n						
average	10.495	31.742	90.566	288.614	846.604	2794.39
empty slot						
deviation	3.64654	8.09475	13.598	23.5842	40.8088	73.0578
average relative	3.49833	3.1742	3.01887	2.88614	2.82201	2.79439
lower bound - 95%	3.42299	3.12403	2.99077	2.87152	2.81358	2.78986
upper bound - 95%	3.57367	3.22437	3.04696	2.90076	2.83044	2.79892
lower bound - 99%	3.39932	3.10826	2.98194	2.86693	2.81093	2.78844
upper bound - 99%	3.59735	3.24014	3.05579	2.90535	2.83309	2.80034
HNO-uknown-n-mod						
average	6.067	24.404	78.132	264.983	798.408	2665.04
empty slot	1.373	4.265	13.308	44.71	133.125	444.692
deviation	2.4171	4.22073	7.51605	14.2169	24.3065	44.2512
average relative	2.02233	2.4404	2.6044	2.64983	2.66136	2.66504
lower bound - 95%	1.9724	2.41424	2.58887	2.64102	2.65634	2.6623
upper bound - 95%	2.07227	2.46656	2.61993	2.65864	2.66638	2.66778
lower bound - 99%	1.9567	2.40602	2.58399	2.63825	2.65476	2.66144
upper bound - 99%	2.08797	2.47478	2.62481	2.66141	2.66796	2.66864
Hybrid-estimated-n-mod						
average	7.914	25.834	73.712	231.299	672.354	2207.39
empty slot	3.199	7.527	19.553	60.995	179.508	592.672
deviation	2.7612	5.22281	9.0099	17.9268	35.1298	81.0544
average relative	2.638	2.5834	2.45707	2.31299	2.24118	2.20739
lower bound - 95%	2.58095	2.55103	2.43845	2.30188	2.23392	2.20237
upper bound - 95%	2.69505	2.61577	2.47568	2.3241	2.24844	2.21241
lower bound - 99%	2.56302	2.54085	2.4326	2.29839	2.23164	2.20079
upper bound - 99%	2.71298	2.62595	2.48153	2.32759	2.25072	2.21399
Hybrid-estimated-n-mod-reduced						
average	7.68	26.396	76.286	235.802	679.149	2218.78
deviation	2.49275	5.20291	10.7513	23.2554	48.5578	128.642
average relative	2.56	2.6396	2.54287	2.35802	2.26383	2.21878
lower bound - 95%	2.5085	2.60735	2.52065	2.34361	2.2538	2.21081
upper bound - 95%	2.6115	2.67185	2.56508	2.37243	2.27386	2.22675
lower bound - 99%	2.49231	2.59722	2.51367	2.33908	2.25064	2.2083
upper bound - 99%	2.62769	2.68198	2.57206	2.37696	2.27702	2.22926