# Connection Establishment in High-Speed Networks

Israel Cidon, *Senior Member, IEEE*, Inder S. Gopal, *Fellow, IEEE*, and Adrian Segall, *Fellow, IEEE*

*Abstract*—**Protocols for establishing, maintaining, and terminating connections in packet-switched networks have been studied in the literature and numerous standards have been developed to address this problem. In this paper, we reexamine connection establishment in the context of a high-speed packet network, introduce a protocol for connection establishment/takedown that is appropriate for such a network, and explain its advantages over previously proposed protocols. The main features of the proposed protocol are: Fast bandwidth reservation in order to avoid as much as possible reservation conflicts, guaranteed release of the reserved bandwidth even under nodal and link failures, soft recovery from processor failures which allows the maintenance of existing connections under processor failure provided the switch and links do not fail.**

**The underlying model that we use is the recently developed PARIS/plaNET network, but the protocol can be adapted to other fast packet networking architectures.**

## I. Introduction

THE advent of fiber-optic media has pushed the transmission speed of communication links to over a Gb/s, representing at least a four-order-of-magnitude increase over typical links in traditional networks. This has not been matched by a corresponding increase in processing capacity of the communication nodes. Thus, processing has become the bottleneck in today's communication systems. This has lead to the development of dedicated hardware to perform very fast packet switching [1]–[3]. In traditional packet-switched networks, the packet handling tasks are all performed by a single general-purpose processor. In fast packet networks like PARIS [2], plaNET [4], and ATM [5], there is a dedicated hardware switch that has the ability to route packets without involvement of a general-purpose processor. Specifically, a network node consists of two parts (see Fig. 1): the switching subsystem (SS), which is a fast but relatively limited function hardware switch, and the Network Control Unit (NCU), which is a slower but more sophisticated processor. All packets that must be relayed through the node travel directly through the hardware in the SS. Only packets that require more complex processing are forwarded to the NCU processor. The assumption is that the time required for switching, that can be handled through hardware, is several orders of magnitude smaller than the time for processing, which requires the capabilities of a general purpose processor.

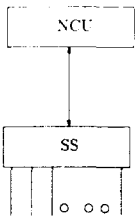



Fig. 1. Node structure.

It is shown in [6]–[9] that several commonly executed control functions like topology maintenance, leader election, and directory functions can be performed substantially faster in this new network than in conventional packet-switched networks (e.g., ARPANET). In this paper, we show that similar results can be obtained for call establishment. Protocols for call establishment have been proposed in several papers and standards documents (e.g. [10],[11]). Here, we construct a call setup protocol that exploits the fast switching hardware and can be used for call establishment with bandwidth reservation. Note that as we are performing packet switching, this "reservation" is not a physical reservation of a time slot. It consists simply of telling the relevant NCU's that some of their local link capacity [12] is being used. This information is used by the NCU's in determining their link loading in order to perform the check described for subsequent calls. We will briefly describe some of the key features of the protocol.

We assume that the call request arrives at a source with a specified destination and bandwidth requirement [12]. Via suitable topology and route determination algorithms, the NCU at the source determines a route through the network for the call and then employs the call setup protocol specified in this paper to establish the call. While the main issue of this paper is the call setup and bandwidth reservation mechanisms and their properties, there is a strong relationship between these mechanisms and the route computation/selection methods employed in the networks. This paper assumes that prior to the call setup and bandwidth reservation attempt, a complete route is precalculated and is known at the call originating node. In other words, we assume that the network employs *source routing*. In general source routing systems, a particular path can be computed via a distributed network wide procedure or alternatively can be computed locally using a topology database which is updated and maintained distributively and dynamically. Our setup/reservation mechanisms can be used in both cases and do not assume a particular method of the source route computation. In the plaNET network [4],

Manuscript received September 1992; revised April 1993; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor J. Turner.

I. Cidon is with the IBM T.J. Watson Research Center, while on leave from the Department of Electrical Engineering, Technion, IIT, Haifa, Israel. (email: cidon@watson.ibm.com)

I.S. Gopal is with the IBM T.J. Watson Research Center, Yorktown Heights, NY. (email: gopal@watson.ibm.com)

A. Segall is with the Department of Computer Science, Technion, IIT, Haifa, Israel. Part of this work was performed when he was visiting the IBM T.J. Watson Research Center. (email: segall@cs.technion.ac.il)

IEEE Log Number 9212249.

the approach used is of a topology database that is updated every time a topological change or a significant utilization change occur in the network. The advantage of this approach is complete and flexible control over the routing characteristics [13], [14]. Using this approach, it is easy to apply various constraints on the computed path (cost, delay, loss, security, etc.) as well as complex policies (some users are not permitted to cross certain links or areas). It is also easy to compute alternate paths of certain qualities and relationships to the primary paths, to apply priorities and preemption mechanisms, etc. The advantage of such local route computations is that they are inherently centralized and can take into account at all times global information. Global information must be maintained by appropriate information update protocols, which can induce large overhead if one attempts to follow too closely rapidly changing patterns. However, many of the considered parameters like policy, cost, and security tend to be static, and the others should be updated at a rate that does not overload the network. Such features can probably be mapped in a limited way to distributed computation methods; however, it will be quite complex and much less flexible to potential changes. For example, if there exist $n$ types of users and each type may use a different set of network resources or require different cost measures, one should develop a distributed method to compute $n$ different shortest paths at the same time. This may be too much, given a plausible scenario that some user types are very rare. The disadvantage of maintaining topology maps is usually the scalability to large networks. Several solutions which involve clustering or centralized/distributed approach combination appears in the literature [15]. As mentioned, the setup/reservation mechanisms of this paper apply to both centralized and distributed source route computation.

On the other hand, our algorithm is not suitable for the case where the routing and reservation processes are combined and performed hop-by-hop via local information. Such a possible solution will be to forward the reservation request over the "best" outgoing link and make the routing decision and reservation at the same time. While such a method has some advantages, e.g., bypass of blocked links, it has many disadvantages as well. The bandwidth reservation action captures costly resources from the network. Therefore, reservations done for a call which is eventually blocked is a pure waste of bandwidth, which becomes significant if there are many competing calls when the network is highly loaded. In such cases, early rejection of calls that have very small chance to be eventually admitted is a positive feature. Note that, in the nonsource routing case, there are two options: use a topology/utilization database to compute the best next path (such as in the ARPA network [16]), or the route can be defined via a local exchange Bellman/Ford type of algorithm [16]. The pros and cons of the two options have been already discussed.

We return now to the call establishment protocol introduced in this paper. The protocol first performs, along the path, a "reservation" of capacity for the call being established. This reservation essentially informs every NCU along the path to set aside the desired capacity for this call. It then performs a "verification" by checking that all NCU's on the path have had enough free resources to reserve the capacity for this call. The verification stage is necessary because every route determination protocol is subject to delays, during which some of the links of the route may become congested. Only if the answer of the verification is affirmative is the call established and exchange of information packets commenced. If the answer is negative, the call must be denied (and potentially reattempted) as no bandwidth is available over the requested route. The protocol ensures that all intermediate nodes have the correct information and that no resource deadlock exists. For example, when a call is not completed, say because of inadequate capacity on some link, the call is disconnected, or a failure is encountered, the bandwidth reservation for that call is removed from the relevant NCU's.

A key feature of the protocol is the fact that the bandwidth reservation is performed very fast, thereby reducing its susceptibility to unnecessary call blocking. A commonly encountered situation in heavily loaded reservation-based systems is a resource contention, whereby several call attempts are made at the same time. Each call is able to reserve some but not all of its required links, causing every call to be blocked even though there is enough capacity in the network to satisfy a subset of the calls. A possible solution to this problem is a centralized bandwidth management approach, together with the usual drawbacks of centralized systems. The approach that we take here is to make the reservation stage as fast as possible, reducing the possibility of contention but not eliminating it completely (note that the critical parameter here is rapid reservation and not fast connection setup). It is easy to realize that if the reservation period is made very short, the probability that another independent call will attempt to "grab" a common resource during the same time will become very small. The reservation period is, thus, the potential conflict period similar to the effect of the signal propagation delay in CSMA [17] systems. In particular, we do not require the NCU's to relay the reservation message. Instead, we use the hardware routing capabilities to ensure that the message is delivered to every intermediate NCU at a maximum speed. Another important parameter is the frequency of utilization updates, since outdated information may increase the probability of blocking. This issue has to be addressed by the designers of the update protocols.

There are also some critical timing issues in the interplay between reservation and data transmission. It is important that both ends begin data transmission only *after* all intermediate NCU's have performed the reservation and terminate data transmission *before* any NCU on the path releases the reservation. Otherwise, transmission conflicts with other connections may cause excessive congestion and, hence, information loss. In addition, since information packets travel only through the switching subsystems, failure of an NCU after call establishment should not normally trigger call cancellation. However, this means that in order to avoid congestion due to overutilization of the capacity, the NCU must learn after recovery about the existence of all calls traversing it before permitting new calls to be established. It is shown formally that the protocol developed in this paper satisfies the important timing and exception requirements.

Another important contribution of the paper is the design of the capacity check process. The period of the capacity check dictates the delay between the reservation and the information transmission in case the check result is positive, and the delay between the reservation and the resource release in case the check result is negative. This basically dictates the reservation "overhead," i.e., the length of the time period in which the common resource is "locked" but not used. A sequential capacity check (hop-by-hop along the call path) would require time that is proportional to the number of nodes along the path. By making use of the hardware routing capabilities, we are able to reduce this to a time that is proportional only to the logarithm of the number of nodes in the path.

To illustrate the potential speedup in reservation time, we present a typical numerical example. In networks such as SNA and TCP/IP, a switched setup or control packet will typically incur thousands of instructions. In addition to the processing itself, such a message may be queued before it is processed (in addition to the queueing delays over the links). Therefore, a typical processing and queueing delay value can be of several milliseconds. Assuming a 5 ms delay, a five-hop route results in an average processing and queueing delay of 25 ms. Switching, transmission, and link queueing delays for control packets are considerably shorter in high-speed networks. If we assume a 1 Gb/s link and 1,000 bits/packet (a very large value for a control packet), we get a transmission delay of $10^{-6}$ seconds per node. Even under a heavy load assumption, as control packets usually travel with high priority, processing will be at least three orders of magnitude higher. Propagation delay varies because of the different distances between users. In most cases, it will also dominate switching transmission and link queueing. For a 10 km path, it is usually assumed to be on the order of 0.05 ms; for a 5,000 km path around 25 ms. This means that for coast-to-coast calls, our improvement in reservation time will be only by a factor of 2 but, for local and medium distance calls (which are prevalent in wide-area networks) the gain is of 2-3 orders of magnitude. We stress again that these calculations refer to the reservation time and not to the total call setup time.

The paper is structured in the following fashion. In the next section, we describe the model, the assumptions, and the requirements. In Section III, we describe the basic call setup protocol, motivate the design choices, and justify them. In Section IV, we formally state and prove the correctness of the protocol. In Section V, we describe the logarithmic capacity check and show how it can be used to improve the basic protocol.

## II. The Model

Each node in the communication network consists of high-speed switching hardware [Switching Subsystem (SS)], which is attached to the communication links, and a single processor [Network Control Unit (NCU)] (see Fig. 1). As mentioned, the switching functions that can be performed in the SS are much faster than functions that require the involvement of the NCU. However, as the SS is a dedicated special-purpose hardware structure, it is not flexible enough to perform higher-level functions and is limited to simple and mainly pipelined intermediate routing functions.

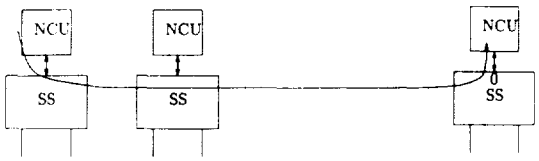



Fig. 2. Automatic network routing.

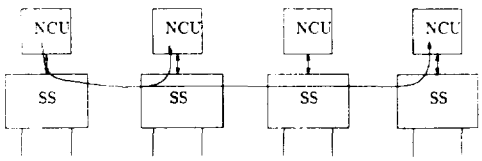



Fig. 3. ANR with selective copy..

The routing of control information is performed using Automatic Network Routing (ANR), sometimes called source routing (note: other routing modes are also supported in various systems such as label swapping and multicast [4]; however, they require a setup phase for their operation and, consequently, are not suitable for the setup phase itself). Each link has an ID assigned to it that is unique within the SS, and the NCU is assigned ID = 0 in each SS. A link may have different ID's at each of its end points. To illustrate the ANR mechanism, assume that a NCU in a certain node wishes to send a packet to an NCU in a certain destination node. It computes a suitable path to that destination node and prefixes the data with a string that is composed of the concatenation of all the link ID's along the computed path. The message travels through the SS's along the predetermined route to the specified NCU. As part of the ANR mechanism we allow the possibility of selected copy to the NCU (see Fig. 3). This is done by adjoining to each of the ID's in the prefix an additional bit, the copy bit. If this bit is set, the packet is copied to the NCU in addition to being forwarded on the appropriate link.

Two types of control messages are used in the protocol described in this paper. One type is similar to an end-to-end information message. These messages employ the ANR technique and propagate end to end through the switching subsystems only, possibly being copied by the NCU's along the path. We shall refer to this type of message as a *transmitted* message. Control messages of the second type are exchanged between neighboring NCU's. We shall refer to this type of message as a *sent* message. Between every pair of neighboring NCU's, there is a Data Link Control (DLC) protocol that ensures data reliability [18] so that *sent* messages are guaranteed to arrive correctly unless the link or the NCU fails. Transmitted messages are not protected by a DLC and, therefore, are not guaranteed to arrive. They may be lost because of congestion in the queues. However, we assume that the buffers in the queue at the NCU are large enough to accommodate all copied messages so that there is no loss of a message that has been copied from the SS to the NCU. Moreover, we shall assume that there is a time interval $T'$, such that if a transmitted message for which the protocol dictates

immediate acknowledgment is not acknowledged within $T'$; either the message or the acknowledgment was lost. Another assumption used in the paper is that there is an integer $K_s$ such that at least one round-trip transmission (message + ACK) out of $K_s$ consecutive transmission trials does arrive correctly with very high probability. The value of $K_s$ depends on the packet loss probability in the network. For example, analysis for the PARIS system shows a packet loss probability of less than $10^{-6}$ per node [19]. (Note: In fast packet switching or ATM, the main source for packet loss is buffer congestion.) All information packets are *transmitted* and, hence, their propagation time is also bounded by $T'$. The timers at all NCU's are set to some interval $T$, where $T \geq T'$. (We assume that the same value of $T$ is used by all NCU's; however, it is easy to accommodate small differences in the timer values.) Call ID's are assumed to be unique networkwide, and all messages related to the same call contain this ID.

This model corresponds to the PARIS/plaNET architecture [4]. Our protocol can, however, be adapted to other architectures like ATM [5]. The relevant layers in the ATM architecture are the ATM layer, AAL (ATM adaptation layer), and the DLC and network layers. Since the switch handles ATM cells, the actions required from the SS in PARIS/plaNET cannot be performed by the ATM switch itself and will have to be done just before the AAL layer. Still, a significant speedup of the reservation process can be achieved compared to protocols that would require the SETUP message to go all the way up to the network layer.

## III. The Call Setup Algorithm

### A. Informal Description

We start by first describing the operation of the protocol when there are no failures on the call path (see Fig. 4). We will then describe how to deal with NCU and link failures that occur during setup or normal operation. We will focus on a particular call and assume that all messages that correspond to a certain call carry a unique identifier which is associated with any call. (The source node which originates the call is responsible for assigning this identifier.)

In normal operation (after the call was initiated and before it is taken down), the source node transmits every $T$ seconds to the destination node a REFRESH message $<A.7.6>$[1] that is acknowledged by the destination with an ACK message $<C.4>$. The REFRESH's and ACK's are copied to the NCU's and serve to inform the source node that the call path and destination node are operative and inform the intermediate NCU's that the call is up at the source and destination. Recall that the variable $K_s$ indicates that, out of $K_s$ consecutive transmissions, at least one round-trip transmission is successful with high probability if the links are up. If the timer at the source expires $K_s + 1$ times without receiving an ACK, the source assumes that the path or destination is inoperative and drops the call $<A.7.3>$ (if the timers at different nodes run at slightly different rates, more than $K_s+1$ timer expirations will be required). If the timer at the destination expires $K_s+2$ times

[1] The notation <> points to the respective line in Section III-B-2.

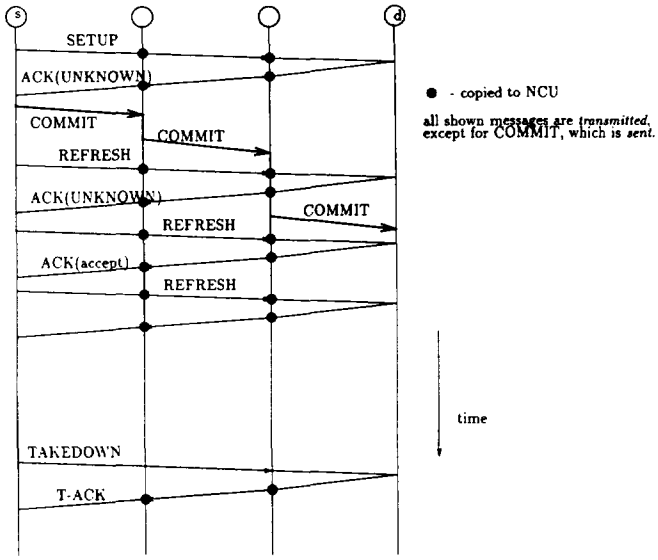



Fig. 4. Setup/takedown example.

without receiving a REFRESH, the destination drops the call $<C.8.7>$. If the timer at an intermediate NCU expires $K_s + 4$ times without the NCU receiving an ACK or REFRESH, the NCU drops the call. These values have been selected to ensure that the source and destination drop the call at a sufficient time before intermediate NCU's, so that information packets cannot arrive at any NCU after the time the latter drops the call and releases the reserved capacity for that call. In this way, we guarantee the required properties as stated in Section I.

The setup of a call uses two types of control messages: SETUP and COMMIT. The SETUP message is identical to the REFRESH message, except that it is transmitted only before the source receives the first acknowledgment. When an intermediate NCU receives a SETUP for the first time for a call, it checks whether there is enough residual capacity to handle the call and reserves the required capacity if the answer is positive $< B.1.6 >$. The destination NCU uses a similar algorithm except that, instead of checking the residual capacity on the forward link, it checks whether the end user is available. In addition, as with the REFRESH message, every SETUP message is acknowledged by the destination NCU by transmitting an ACK to the source. When the timer expires, the source retransmits a SETUP message unless it had already received at least one ACK. After having received the first ACK, it sends (recall the difference between send and transmit) COMMIT and starts transmitting REFRESH at timer intervals as explained before. The purpose of the SETUP message is to inform the NCU's on the path of the attempt to establish the call and to have them reserve the capacity. Since transmitted messages propagate quickly, the SETUP ensures fast reservation, thereby minimizing the probability of conflicts. In addition, it ensures maximum parallelism in the distributed process of checking for sufficient residual capacity over the entire path.

The source sends a COMMIT message immediately after receiving the first ACK. The purpose of the COMMIT is to find out whether all NCU's have indeed been able to

reserve the capacity for the call. When it receives a COMMIT message, if capacity is reserved, an intermediate NCU sends a COMMIT message to the next NCU. Otherwise, it sends an ABORT message and the latter is propagated on the path $< B.2.6 >$. Both COMMIT and ABORT are *sent* messages, i.e., they propagate hop-by-hop using error detection and retransmission, thus ensuring reliable communication. If every NCU on the path has reserved the capacity for the call, the destination will receive COMMIT; otherwise, it will eventually receive ABORT. Note that while there may be several SETUP messages transmitted, for a given call there is exactly one COMMIT/ABORT chain.

For the purpose of our description, we shall say that the NCU is in *up* state for a call when the setup procedure is completed as far as the NCU is concerned. After receiving SETUP, an NCU is in an intermediate state called *pending*. While in *pending*, the variable *reserved* indicates whether the capacity has been reserved for that call or not. In *up* state, the *reserved* variable is always *yes*. It will also be convenient to indicate by a special state the situation when an NCU knows nothing about a call. For this case, we distinguish between the destination and intermediate NCU's on one hand and the source NCU on the other. When the destination or intermediate NCU knows nothing about a call, we shall say that the call is in the *nonexistent* state at the NCU. This can happen if the initialization procedure had not been started as far as the NCU knows, if the call had been cancelled at that NCU or, by definition, if the NCU is nonoperational. If the call had not been initialized yet, the source NCU is said to be in the *nonexistent* state for that call. After leaving nonexistent state, the call cannot return to this state at the source. Rather, if the call is cancelled or the source NCU fails, we say that the call enters the *finished* state. Afterwards, by definition, the call stays in the finished state forever. The reason for this state is to ensure that the identifier of a call is never reused. In practice, the identifier may be reused a sufficiently long time after the corresponding call entered the finished state.

As said before, the destination transmits ACK in response to every SETUP or REFRESH. There are three types of ACK messages, depending on the current state of the destination. If the destination is in the *up* state, i.e., after having received COMMIT, it uses ACK(accept). If the destination is in the *pending* state, i.e., after the SETUP but before the COMMIT/ABORT, and has reserved the capacity for the call, it uses ACK(unknown). If the state is *pending*, but the check for capacity and user availability turned out negative or if at any time the destination wishes to take down the call, it uses ACK(reject). An ACK(reject) need not be copied to the intermediate NCU's because, if it does arrive at the source, it will cause the drop of the call. The other two types are copied and used as described earlier. In addition, ACK(accept) is used to bring the intermediate NCU to state *up*. When the source receives the first ACK(accept), it makes a transition to the *up* state and can start transmitting data messages to the destination. If it receives an ACK(reject) or wishes to take down the call, it drops the call and transmits a TAKEDOWN message, to which the destination responds with a T-ACK. The latter is copied to every NCU and causes it to drop the call. The TAKEDOWN T-ACK pair is used only to accelerate the release of the reserved capacity; $K_s + 4$ periods later, the intermediate NCU's and the destination would have dropped the call even without the TAKEDOWN. (Note that, at that time, call identifiers may be reused but we omit this function to facilitate the explanation.)

Until now, we have considered no topological changes. In the following, we shall describe the extensions to the protocol to handle link and NCU failures and recoveries. Since for sent messages we do not have a bound on the delay in the NCU, the COMMIT chain propagation cannot be protected by a timeout. Consequently, a source that receives messages of ACK(unknown) must wait until it receives a message of ACK(accept) or ACK(reject) in order to decide upon the fate of the call. This will happen only after COMMIT/ABORT reaches the destination. In order to avoid an indefinite wait in the case of a link or NCU failure that blocks the propagation of the COMMIT/ABORT, we use the fact that if a message is sent between neighboring NCU's, the DLC ensures that (in finite time) either the message is received correctly or a failure is detected. When it detects a failure, an NCU in the *pending* state sends back a CNCL message. Similarly, when the COMMIT/ABORT chain reaches an NCU whose forward link or NCU is nonoperational, the NCU sends back a CNCL message. The purpose of the CNCL message is to ensure that the source will drop the call in the case when the COMMIT/ABORT chain is interrupted. The CNCL is sent on the reverse path until it reaches the source or a failed link, a failed NCU, or an NCU in the *up* state. If it reaches the source, the latter drops the call and transmits TAKEDOWN. If it reaches another failure, then it is discarded but the NCU on the other side of the failure must have initiated, or will initiate, another CNCL. The situation when it reaches an NCU in the *up* state is when the CNCL is sent by an NCU in the *pending* state, is delayed in the queue at the sending NCU, and by the time it reaches the next NCU backward an ACK(accept) had traversed the link connecting the switching systems of the two NCU's. However, this means that the destination was in the *up* state, so the COMMIT chain was not interrupted and has reached the destination. Consequently, the CNCL chain can be terminated at an NCU in the *up* state. Because of this situation, intermediate NCU's do not drop the call upon receiving CNCL since they are not sure that the call will indeed be dropped at the source. If intermediate NCU's would have dropped the call upon receiving CNCL, then the property that a link is never used beyond its capacity (Lemma 7) might have been violated.

The last problem to be solved is that of NCU's that recover from failures. Note that, in our model, an NCU failure does not imply the failure of its adjacent links, thus calls that have already been established should not be disrupted. However, in a failure, the NCU may lose its memory including the information regarding existing calls and the residual capacity on its links. In order to prevent link overutilization that may cause excessive packet loss, the recovered NCU cannot accept new calls until it restores this information. The way to accomplish this is to require the recovering NCU to enter an *exception* mode, where it records information about existing traversing calls from copied ACK messages but does not allow

new calls to be set through it. The duration of the exception mode is $K_e$ intervals, where $K_e = K_s + 4$ (the algorithm is correct for any $K_e \geq K_s + 4$, but the exception mode should obviously be made as short as possible). The operation of the intermediate NCU in exception mode is as follows. Blocking of new calls in exception mode is done by setting *reserved* = no if SETUP or COMMIT arrives and by transforming any COMMIT chain into an ABORT chain <$B$.1.4>, <$B$.2.6>. If an ACK(accept) arrives for a call, this means that the call is up at the destination NCU and the COMMIT chain had succeeded for this call in the past. In this case, the state is set to *up* and *reserved* to *yes*. Receipt of a REFRESH or ACK(unknown) in exception mode puts the NCU in an ambiguous situation, so that *reserved* can be set neither to *yes* nor*no*. It cannot be set to *yes* because this may be a call for which COMMIT has not been received yet, and in exception mode the NCU does not know if it has enough free capacity for it. It cannot be set to *no* because the COMMIT may have traversed the NCU before the failure, but may not have yet reached the destination NCU. In this case, if the COMMIT chain is very slow and the NCU leaves exception mode in the interim, it may accept new calls without taking into account the load of the considered call. The solution is to set the *reserved* variable to a third value *maybe*, whose meaning is that the call should be taken into account in terms of load, but the COMMIT should be transformed into ABORT if it arrives later.

An exception period is defined at the destination as well, in order to ensure that all calls that had existed before the failure will be dropped by the source. No ACK's are transmitted in this period.

As a final remark, we note that if a link failure occurs, the adjacent NCU retains all calls that traverse that link until they are dropped by the regular timeout mechanism. This ensures that no congestion occurs if the link comes up and the failure was not detected by one of the end points.

### B. Formal description

#### 1) Variables and Messages

Variables at source NCU
- per adjacent link
  - used-capacity = total capacity of calls in *pending* or *up*
- per call
  - state = nonexistent, pending, up, finished
- per call, for calls in pending or up
  - counter = 0,1,2,...,$K_s$
  - timer = starts when state ← pending, expires every T seconds
  - capacity-available = true if for the forward link and the given call, link-capacity minus used-capacity > new call capacity
  - drop = yes, no
  - flag = setup, finish

Variables at intermediate NCU's
- node variables (common to all calls)
  - mode = exception, normal
  - exception-counter = 0,1,2,..,$K_e$
- per adjacent link
  - used-capacity = total capacity of calls in *pending* or *up* with *reserved* = *yes* or *maybe* traversing the link.
- per call
  - state = nonexistent, pending, up
- per call, for calls in pending or up
  - reserved = yes, no, maybe
  - counter = 0,1,2,...,$K_s + 4$
  - timer = starts when state ← pending, expires every T seconds
  - capacity-available = true if on both backward and forward links, link-capacity minus used-capacity > call capacity

Variables at destination NCU
- node variables (common to all calls)
  - mode = exception, normal
  - exception-counter = 0,1,2,..,$K_e$
- per adjacent link
  - used-capacity = total capacity of calls in *pending* or *up* with *reserved* = *yes* or *maybe* traversing the link.
- per call
  - state = nonexistent, pending, up
- per call, for calls in pending or up
  - reserved = yes, no, maybe
  - counter = 0,1,2,...,$K_s + 2$
  - timer = starts when state ← pending, expires every T seconds
  - capacity-available = true if on backward link, link-capacity minus used-capacity > call capacity
  - user-reachable = true if end user is reachable

Transmitted messages copied to the NCU
- ACK(unknown)
- ACK(accept)
- T-ACK
- SETUP
- REFRESH

Transmitted messages not copied to NCU
- ACK(reject)
- TAKEDOWN

Sent messages
- COMMIT
- ABORT
- CNCL

#### 2) The Algorithm

SOURCE - mode = normal, algorithm per call

A.1 call initialization ( only in *nonexistent*, when capacity-available and when link/NCU forward is operational)

```
A.1.1     counter ← 1
A.1.2     transmit SETUP
A.1.3     state ← pending
A.1.3     counter ← 1
A.1.4     flag ← setup
A.1.5     drop ← no
A.2    receives ACK(type)
             /*state ≠ nonexistent, finished*/
A.2.1     counter ←0
A.2.2     if type = reject, then drop ← yes
A.2.3     if drop = no, then
A.2.4        if flag = setup, then
A.2.5           flag ← refresh
A.2.6           send COMMIT forward
A.2.7        if type = accept, then
A.2.8           state ← up
A.3    receives CNCL
A.3.1     if state = pending, then drop ← yes
A.4    takedown initialized
          ( only in state pending or up )
A.4.1     drop ← yes
A.5    link/NCU forward fails
A.5.1     if state=pending, then drop ← yes
A.6    receives T-ACK
A.6.1     disregard

SOURCE - node algorithm

A.7    timer expires
A.7.1     if state = pending or up, do
A.7.2        counter ← counter + 1
A.7.3        if counter = K_s + 1, then
                drop ← yes
A.7.4        if drop = no, then
A.7.5           if flag =setup, then
                   transmit SETUP
A.7.6           else, transmit REFRESH
A.7.7        else,
A.7.8           transmit TAKEDOWN
                state ← finished
A.8    NCU comes up
A.8.1     if comes up for the first time, then
             for all calls,
             state +← nonexistent
A.9    NCU fails
A.9.1     for calls in pending or up, do
             state ← finished
A.10   data is transmitted only in state = up

INTERMEDIATE NCU- algorithm per call

B.1    receives SETUP
B.1.1     counter ← 0
B.1.2     if state = nonexistent, then
B.1.3        state ← pending
B.1.4        if mode = exception,then
                reserved ← no
B.1.5        else,
B.1.6           if capacity-available, then
                   reserved ← yes
B.1.7           else, reserved ← no
B.2    receives COMMIT or ABORT
                          /* state ≠ up*/
B.2.1     if state ≠ nonexistent, then
B.2.2        if link/NCU-forward is
                operational, then
B.2.3           if mode = normal and
                received COMMIT and
                reserved = yes, then
B.2.3a             send COMMIT forward
B.2.4           else,
B.2.5              reserved ← no
B.2.6              send ABORT forward
B.2.7        else,
B.2.8           reserved ← no
B.2.9           send CNCL backward
B.3    receives ACK(type)  /* type ≠ reject*/
B.3.1     counter ← 0
B.3.2     if state = nonexistent, then
                    /* mode = exception*/
B.3.3        state ← pending
             reserved ← maybe
B.3.4     if state = pending and
                type = accept, then
B.3.5        state ← up, reserved ← yes
B.4    receives CNCL or link/NCU-forward fails
B.4.1     if link/NCU-backward is operational
             and state = pending, then
B.4.2        send CNCL backward
B.5    receives T-ACK
B.5.1     state ← nonexistent
B.6    receives REFRESH
B.6.1     counter ← 0
B.6.2     if state = nonexistent, then
                    /*mode = exception*/
B.6.3        state ← pending, reserved ← maybe

INTERMEDIATE NCU - node algorithm
          (global to all calls)

B.7    timer expires
B.7.1     if mode = exception, then
B.7.2        exception-counter ←
                   exception-counter + 1
B.7.3        if exception-counter = K_e, then
                mode ← normal
B.7.5     for all calls with
                state = pending or up, do
B.7.6        counter ← counter + 1
B.7.7        if counter = K_s + 4 , then
B.7.8           state ←nonexistent
B.8    NCU comes up
B.8.1     mode ← exception
B.8.2     exception-counter ← 0
B.9    NCU fails
B.9.1     for all calls with
```

```
                state = pending or up, do
                  state ← nonexistent

DESTINATION-mode = normal, algorithm per call

C.1     receives SETUP
C.1.1     if state = nonexistent, then
C.1.2       state ← pending
C.1.3       if user-reachable and
                  capacity-available,then
              reserved ← yes
C.1.4       else, reserved ← no
C.1.5     if state = pending, then
C.1.6       if reserved = yes, then
              transmit ACK(unknown)
C.1.7       else, transmit ACK(reject)
C.1.8     else, transmit ACK(accept)
C.1.9     counter ← 0
C.2     receives ABORT          /*state ≠ up*/
C.2.1     if state = pending, then
C.2.2       reserved ← no
C.3   receives COMMIT           /*state ≠ up*/
C.3.1   if state = pending and
          reserved = yes, then
C.3.2     state ← up
C.4   receives REFRESH*   /*state ≠nonexistent*/
C.4.1     if state = pending, then
C.4.2       if reserved = yes, then
              transmit ACK(unknown)
C.4.3       else, transmit ACK(reject)
C.4.4     else, transmit ACK(accept)
C.4.5     counter ← 0
C.5     takedown initialized
              (only in pending or up)
C.5.1     state ← pending
C.5.2     reserved ← no
C.6     receives TAKEDOWN
C.6.1     state ← nonexistent, transmit T-ACK

DESTINATION - mode = exception
            algorithm per call
C.7     ignore all messages

DESTINATION - node algorithm
                  (global to all calls)
C.8     timer expires
C.8.1     if mode = exception, then
C.8.2       exception-counter ←
              exception-counter + 1
C.8.3       if exception-counter = K_e, then
              mode ← normal
C.8.4     else, for all calls with
              state = pending or up, do
C.8.5       counter ← counter + 1
C.8.6       if counter = K_s + 2, then
C.8.7         state ← nonexistent
C.9     NCU comes up
C.9.1     mode ← exception
C.9.2     exception-counter ← 0
C.10    NCU fails
C.10.1    for calls in pending or up, do
            state ← nonexistent
C.11    data is transmitted only in state = up
```

## IV. Proof of Correctness

In this section, we prove that the call setup protocol presented in Section III-B operates correctly despite an arbitrary sequence of NCU failures, link failures, or message corruption. Recall that we say that SETUP, REFRESH, TAKEDOWN and all ACK's are transmitted messages, while COMMIT, ABORT, and CNCL are sent messages. Also, recall that we assume that after a control message is transmitted by the source, its corresponding ACK arrives at the source within $T$ seconds or the message or ACK is lost.

*Lemma 1:* **General properties**

Suppose that the source starts a call initialization $<A.1>$ at time 0, say. Then,

1. At the source NCU, a call may leave nonexistent state only once. It enters pending state, then possibly up state and then finished state. After entering finished state, it never leaves it.
2. It will transmit at most $K_s - 1$ SETUP's, followed by 0 or more REFRESH's, followed by at most one TAKEDOWN, at times $0, T, 2T, 3T, \ldots$. An ACK is transmitted by the destination NCU only upon receipt of a SETUP or REFRESH. A T-ACK is transmitted by the destination NCU only upon receipt of the TAKEDOWN. If an operational intermediate NCU does not receive the SETUP, then no NCU following it can receive it either and no ACK is transmitted. If an ACK or T-ACK is not received by an operational intermediate NCU, it is not received by any NCU preceding it.
3. At the destination NCU, a call may leave nonexistent state only once. It may enter nonexistent state after having been in pending or up state only once. This happens if it is dropped, due to timer $< C.8.7 >$, TAKEDOWN $< C.6.1 >$, or when the NCU fails $< C.10>$. After that time, it never leaves nonexistent state again.
4. If an intermediate NCU drops the call because of timer $<B.7.8>$ at time $t$, then the call is in finished state at the source NCU at time $t - 2T$ and forever thereafter and the destination NCU is in nonexistent state for that call at time $t - T$ and forever thereafter.
5. At an intermediate NCU, a call may enter nonexistent state after having been in pending or up state if it is dropped, due to timer $<B.7.8>$, T-ACK $<B.5.1>$, or when the NCU fails $<B.9.1 >$. If it is dropped, it never leaves nonexistent state again. If it enters nonexistent state when the NCU fails, it may leave it again and reenter pending state.
6. There is exactly one COMMIT/ABORT chain per call. It starts at the source node and ends either at the destination

node or at an NCU with link-forward or NCU-forward down.

7. All claims in the algorithm are correct.

*Proof:* The source NCU algorithm never allows it to leave finished state once entered for a given call [hence, a)]. Part b) is obvious from the algorithm. Next, we prove c). In order to leave nonexistent state, the destination NCU must receive a SETUP message in normal mode. We want to show that, if at time $t$ the destination enters nonexistent state after it had been in pending or up, it will never receive a SETUP message after $t$ for that call while in normal mode and, in fact, it will never receive in normal mode any control message transmitted by the source for that call. If at time $t$ the destination receives TAKEDOWN, then [by b)] after it has transmitted the TAKEDOWN, the source never transmits any message for this call. Consequently, because of FIFO on the source/destination path, the destination will never receive another message transmitted by the source (note that COMMIT and ABORT may arrive, but those are not transmitted messages). Next, consider the case when at time $t$, a failure at the destination causes the call to enter nonexistent state (from pending or up). Upon recovery, the destination enters exception mode, where it stays for $K_eT$ and does not react to messages $<C.7>$. Therefore, at least between $t+T$ and $t+K_eT$, the source receives no ACK messages. Since the destination NCU is in pending or up before $t$, the source has also been in these states at some point before $t$. Also, since $K_e - 1 > K_s$, the timer expires at the source $< A.7.3 >$ at the latest at time $t + K_sT$, provided that the source has not already entered finished state before. Hence, if the source is not in state finished already, it will transmit to this site no later than $t + K_sT$. Therefore, the last possible control message belonging to this call is transmitted by the source at time $t + K_sT$. Consequently, the destination cannot receive any such message after $t + (K_s + 1)T$, which is before $t + K_eT$, the earliest time when the destination may be in normal mode. Finally, consider the case when at time $t$ the destination NCU enters nonexistent state from pending or up due to the timer $< C.8.7 >$. This means that during $(t - (K_s + 2), t)$, the destination has received no SETUP's or REFRESH's and has transmitted no ACK's. Consequently, at least during $(t-(K_s+1)T, t)$, the source has received no ACK's. Hence, the timer at the source expires $< A.7.3 >$ at the latest at time $t - T$, if the source had not already entered state = finished. Therefore, the last possible control message belonging to this call is transmitted by the source at time $t - T$. Consequently, the destination cannot receive any such message after time $t$.

To prove d), note that if an intermediate NCU drops the call in $< B.7.8 >$ at time $t$, then it receives no ACK and no REFRESH or SETUP during $(t - (K_s + 3)T, t)$ although it had been operational for this period and had the call in pending or up. This means that the source has previously initialized the call and does not receive an ACK at least during $(t - (K_s + 2)T, t)$. Consequently, if the source does not previously enter finished state, the timer at the source will expire $< C.8.6 >$ before or at time $t - 2T$, causing it to enter finished state. Similarly, since the intermediate NCU receives no REFRESH or SETUP during $(t - (K_s + 4)T, t)$, the destination NCU receives no REFRESH or SETUP at least during $(t-(K_s+3)T, t)$. Consequently, if the destination does previously enter nonexistent state, the timer at the destination will expire $< C.8.6 >$ at the latest at time $t - T$, causing it to enter nonexistent state.

Next we prove e). If the intermediate NCU drops the call due to timer, by d) it will never receive a transmitted message or ACK for that call afterwards, so it cannot reenter pending state. Similarly, after receiving T-ACK, it cannot receive any transmitted message or ACK for that call. However, if the intermediate NCU fails, the call may remain operational between the source and the destination, and in this case the NCU will reenter pending state after it recovers $<B.3.3>$.

We note that f) is obvious from the algorithms.

Next, we prove g). The claim in $<A.2>$ is obvious. Since there is only one COMMIT/ABORT chain and COMMIT brings the NCU to *up*, claim $< B.2 >$ follows. ACK(reject) is not copied to the NCU, hence claim $<B.3>$.

Next, we prove the claim in $<B.3.2>$. Suppose the contrary, i.e., an intermediate NCU receives an ACK in normal mode in state=nonexistent, at time $t_1$ say. Let $t_2, t_3$ be, respectively, the times when the message causing the considered ACK was transmitted by the source and was received by the intermediate NCU. We have $t_1 - T < t_2 < t_3 < t_1$. Note also that at $t_2$, the source is in pending or up for that call. Let $t_4$ be the last time before $t_1$ when the NCU entered state=nonexistent for that call. Note that the event at $t_4$ cannot occur because of receipt of a T-ACK— this would mean that TAKEDOWN was transmitted by the source before $t_2$, causing the source to enter finished state, contradicting the fact that the source is in pending or up at $t_2$. Note, also, that the event at $t_4$ cannot occur in $< B.7.8 >$ (timeout) because, by Lemma 1c), this means that the source drops the call before $t_4 - 2T$, hence before $t_2$. This contradicts, as before, the fact that the source is in pending or up at $t_2$. The only other possibility is that, at $t_4$, the NCU fails. However, since the NCU is in normal mode at $t_1$ and an NCU stays in exception mode $K_eT$ seconds after coming up, this means that $t_4 \leq t_1 - K_eT < t_1 - (K_s + 4)T)$. However, the NCU receives no ACK during the time interval $(t_1 - (K_s + 4)T, t)$ because such an ACK would take it out of nonexistent state, contradicting the fact that the NCU is in nonexistent state from $t_4$ to $t_1$. By the same argument as in the proof of d), this implies that the source must drop the call before $t_2$, contradicting the fact that the call is in pending or up at $t_2$ at the source.

Next, we prove the claim in $<B.6.2>$. Suppose the contrary, i.e., an intermediate NCU receives a REFRESH in normal mode in state=nonexistent, at time $t_1$ say. Let $t_2$ be the time when the REFRESH was transmitted by the source. We have $t_1 - T < t_2 < t_1$. Note, also, that at $t_2$ the source is in pending or up for that call. Let $t_4$ be the last time before $t_1$ when the NCU entered state=nonexistent for that call. Note that the event at $t_4$ cannot occur because of receipt of a T-ACK because this would mean that TAKEDOWN was transmitted by the source before $t_2$, causing the source to enter finished

state, contradicting the fact that the source is in pending or up at $t_2$. Note, also, that the event at $t_4$ cannot occur in $<B.7.8>$ because, by c), this means that the source drops the call before $t_4 - 2T$, hence before $t_2$. This contradicts, as before, the fact that the source is in pending or up. The only other possibility is that, at $t_4$, the NCU fails. However, since the NCU is in normal mode at $t_1$ and an NCU stays in exception mode $K_eT$ seconds after coming up, this means that $t_4 \leq t_1 - K_eT < t_1 - (K_s + 4)T)$. However, the NCU receives no ACK during $(t_1 - (K_s + 4)T, t)$ because such an ACK would take it out of nonexistent state, contradicting the fact that the NCU is in nonexistent state from $t_4$ to $t_1$. By the same argument as in the proof of d), this implies that the source drops the call before $t_2$, again contradicting the fact that the call is in pending or up at $t_2$ at the source.

The claims in $<C.2>$ and $<C.3>$ hold because there is only one COMMIT/ABORT chain and only a COMMIT puts the destination in up state. The claim in $<C.4>$ holds because the source transmits REFRESH only after receiving the first ACK, and the destination transmits an ACK only after leaving nonexistent state. Moreover, if the destination fails and recovers, it stays in exception mode without transmitting any ACK's for sufficient time to force the source to drop the call. □

*Lemma 2* **(Termination)**: If a call setup procedure is started, either both the source and destination will eventually go to *up* or the source will go to *finished* and the destination to *nonexistent*.

*Proof:* Assume the contrary. The only way for this to happen is for the source to forever stay in pending while receiving ACK(unknown)'s at time intervals strictly less than $K_sT$ transmitted by the destination. When it sends ACK(unknown), the destination is also in pending. Since, by Lemma 1, the source cannot transmit SETUP after $K_s$ time periods, the ACK(unknown) are eventually acknowledgments to REFRESH messages. For ACK(unknown) to be transmitted, the REFRESH must find the destination in *pending* with *reserved* = *yes* $<C.4.2>$ (pending-yes, for short). This implies that the destination never receives a COMMIT or ABORT, as COMMIT causes a transition out of the pending state and ABORT sets *reserved* to *no*. Afterwards, there would be no way for the destination NCU to return to pending-yes. Since a COMMIT chain is started by the source when it receives the first ACK, the destination can receive no COMMIT/ABORT because the chain of COMMIT/ABORT terminates at an NCU with link/NCU forward failed $<B.2.12>$. This NCU will send CNCL back and start a CNCL chain. Consider the NCU $i$ nearest to the source that starts a CNCL chain. We first claim that all NCU's closer to the source than $i$ entered pending state and do not leave it. They entered pending state because they received COMMIT/ABORT or SETUP beforehand. They cannot drop the call as a result of receiving T-ACK, because the source does not transmit TAKEDOWN as it continuously receives ACK(unknown). They cannot drop the call because of the timer since then, by Lemma 1, the source would previously drop the call. They cannot leave pending to up, because the destination transmits no ACK(accept). They cannot leave it to nonexistent state, since the only way for an NCU to do that is in a failure— in that case, another CNCL chain is started closer to the source. Consequently, the CNCL chain started at $i$ arrives at the source, causing it to drop the call and provide a contradiction. □

For our discussion henceforth, a link is considered one-directional so that two adjacent nodes are connected by two links, one in each direction. We denote by $NCU_i$ the NCU before link $i$. We state the correctness properties in terms of the following sets.

$U_i$ = set of calls for which data messages may traverse link $i$

$R_i$ = set of calls in pending or up with reserved = yes at $NCU_i$, if $NCU_i \neq$ *source*
= set of calls in pending or up at $NCU_i$ if $NCU_i$ = source

$M_i$ = set of calls in pending or up with reserved = maybe at $NCU_i$ if $NCU_i \neq$ source or destination
= $\Phi$ if $NCU_i$ = source or destination

Note that if $i$ is a link in the forward direction, i.e., from source to destination, a call enters $U_i$ when the source NCU enters state = up for that call and exits $U_i$ at time $T$ seconds after the source enters finished or when the NCU receives TAKEDOWN, whichever comes first. This is similar to a link in the backward direction, with the destination replacing the source and T-ACK replacing TAKEDOWN. Because of Lemma 1, for a given link, a call may enter $U_i$ at most once. We also define the following traffic capacities.

$C(j)$ = capacity of call $j$. $C(X) = \sum_{j \in X} C(j)$.
$C_i$= capacity of link $i$.

Lemmas 2 and 4 demonstrate that a call setup will always be successful under stable network conditions and if adequate capacity is available.

*Lemma 3* **(Guaranteed reservation)**: A call setup initiated for call $j$ will cause $j \in R_i$ for all links $i$ along its path if the following conditions hold.

1. All links and NCU's on the path of the call are operational at initiation time and for $K_s + 4$ timeout periods afterwards.
2. At no time from the initiation of the setup until $K_s + 4$ timeout periods afterwards does there exist either a NCU along the path in exception mode or a link $k$ on the path s.t. $j \notin R_k$ and $C_k - C(R_k \cup M_k) > C(j)$.

*Proof:* Since one of our basic assumptions is that at least one SETUP out of $K_s$ successfully traverse all NCU's on the path, each NCU along the path will receive a SETUP in state=nonexistent in normal mode. Since b) ii) holds, it will put the call in pending state with reserved=yes. □

*Lemma 4* **guaranteed setup**: If the call $j \in R_i$ for all links $i$ along its path (as in Lemma 4 ), if no link or NCU failures ever occur, and no takedown is triggered by either the source or destination, the source and destination will enter state=up for that call and will stay there.

*Proof:* When the first ACK reaches the source, the latter send COMMIT. As no failures occur and every link has the

call reserved, the COMMIT message will pass through to the destination. The destination will enter the *up* state. After this occurs, the assumption of message delivery in a failure-free environment will guarantee that a REFRESH or SETUP will be received by the destination in *up* state, which will transmit an ACK(accept) to be received at the source. The source will then enter *up* state. □

In order to prove the *always reserved* and *no overutilization* properties (Lemmas 6 and 7), we need an auxiliary lemma. For a given link $i$, we define the set $Q_i$ as follows. A call enters $Q_i$ at setup time if reserved is set to yes, i.e., in $< A.1 >$, $<B.1.6>$, or $<C.1.3>$. If $i$ is a forward link, a call exits $Q_i$ when $NCU_i$ receives TAKEDOWN or $T$ seconds after the call enters finished state at the source, whichever comes first. If $i$ is a backward link, a call exits $Q_i$ when $NCU_i$ receives T-ACK or $T$ seconds after the call enters nonexistent state from pending or up at the destination, whichever comes first. Because of Lemma 1, for a given link $i$, a given call may enter $Q_i$ at most once.

*Lemma 5* **Auxiliary**: For every $i$ holds $U_i \subset Q_i$. If $NCU_i$ is in normal mode, then $Q_i \subset R_i \cup M_i$.

*Proof:* The source or destination can enter up state only after all NCU's have received SETUP and have set reserved=yes; hence, at the time a call enters $U_i$, it has already entered $Q_i$. If a call is in $U_i$, the condition for leaving it is the same as for leaving $Q_i$. Hence, $U_i \subset Q_i$.

A call $j$ enters $Q_i$ only if $NCU_i$ is in normal mode and only if reserved is set to yes, hence at the same time $j$ enters $R_i$. Now consider all events that cause $j$ to be out of $R_i \cup M_i$ while $NCU_i$ is in normal mode. We show that, in each, such a case holds $j \notin Q_i$. Call $j$ may leave $R_i \cup M_i$ while $NCU_i$ is in normal or exception mode, at time $t$ say, because of T-ACK $< B.5.1 >$ or because of timer $< B.7.8 >$. In the first case, at time $t$, it also leaves $Q_i$ at $t$. In the second case, because of Lemma 1, it leaves $U_i$ and $Q_i$, at or before $t$.

The only other case when the call $j$ is not in $R_i \cup M_i$ while $NCU_i$ is in normal mode is if $NCU_i$ enters normal mode from exception mode, at time $t$ say, without call $j$ in $R_i \cup M_i$. In this case, $NCU_i$ had been in exception mode during $(t - K_e T, t)$. If at any time during that interval a T-ACK is received or the timer expires (as shown before), the call has left $Q_i$ at or before that time so at $t$ it is not in $Q_i$. Otherwise, the only way for it not to be in $R_i \cup M_i$ is if $NCU_i$ receives no REFRESH or ACK during the entire exception interval (no SETUP's can be received since the source transmits no SETUP's after sending COMMIT). However, this means that no REFRESH is received by the destination and no ACK is received by the source at least during $(t - (K_e - 1)T, t)$. Consequently, they enter nonexistent and finished state, respectively, before $t - T$, implying that $j$ leaves $Q_i$ before $t$.

*Lemma 6* **Always reserved**: For any link $i$ for which $NCU_i$ is in normal mode, $j \in U$ implies $j \in R_i \cup M_i$.

*Proof:* Follows from Lemma 3.

Lemma 5 shows than a link is never used beyond its capacity.

*Lemma 7* **No overutilization**: For every $i$, it holds that $C(U_i) < C_i$.

*Proof:* We show a stronger result: $C(Q_i) \leq C_i$. In order to show that, it is sufficient to show that immediately after a call enters $Q_i$, it holds that $C(Q_i) < C_i$. A call $j$ is placed in $Q_i$ only in normal mode and only if (just before that) it holds that $C_i - C(R_i \cup M_i) > C(j)$. At that time, it is also placed in $R_i$. Consequently, just after the placement takes place, it holds that$C_i > C(R_i \cup M_i)$ and $NCU_i$ is in normal mode. Therefore, from Lemma 3, it holds that $C_i > C(Q_i)$.

Lemma 6 shows that the reserved capacity for any link will eventually reflect the actual usage of that link.

*Lemma 8* **Accuracy of reservation**:

1. There exists no call $j$ and no link $i$ such that $j \in M_i$ forever.
2. There exists no call $j$ and no link $i$ such that $j \in R_i - U_i$ forever.

*Proof:* Call $j$ enters $M_i$ if ACK(unknown) is received in exception mode in state=nonexistent. From Lemma 4, the destination cannot transmit ACK(unknown) forever. Either both source and destination will go to up or the call will be dropped by both source and destination. In the first case, ACK(accept) is transmitted by the destination and received at the source; hence, unless $NCU_i$ fails (in which case $j$ leaves $M_i$), it is also received at $NCU_i$. Then, $j$ leaves $M_i$ and enters $R_i$. In the second case, the call is dropped by $NCU_i$ as well and, hence, leaves $M_i$.

To prove ii), note that if a call is in some $R_i$, there must have been a call setup attempt. By Lemma 4, either the source and destination will eventually go to up or both will drop the call. In the first case, $j \in U_i$. In the second case, $NCU_i$ will also drop the call. □

## V. Extensions

The main goal of this section is to introduce new methods that accelerate the capacity check part of the call setup protocol for long calls. The basic approach taken in previous sections was to use a sequential hop-by-hop capacity check in order to guarantee the capacity availability along the call path. This check must be completed before any user information can flow over this path. Basically, a COMMIT message is forwarded by every NCU along the path as long as this NCU has reserved the requested capacity. If the NCU has not been able to make such a reservation, the COMMIT chain is interrupted and replaced from this point on by an ABORT message. The ABORT message makes its way to the destination with no further change. Once the destination receives the ABORT or COMMIT message, the process is completed. A special mechanism was introduced to guarantee that either this process is terminated (and it will, if no failure occurs) or a CNCL message is received by the source.

It is easy to observe that the capacity check process does not affect the operation of the rest of the algorithm. The only requirement is that, in the absence of failures, it will terminate.

The capacity check algorithm as described previously is inherently a sequential process. If we count each message transmission, reception and processing as a basic delay unit, it introduces a time delay which is linear with the length of the path. In most practical networks, this linear time cost is not critical since the diameter of the network is kept bounded by the topology design because of end-to-end delay considerations. However, in principle if the end-to-end paths are excessively long, this linear time cost may be a delay bottleneck in the process of the call setup.

The capacity check process is not sequential in nature. It is basically a distributed implementation of an AND or OR Boolean function. It is sufficient that a single node has not reserved the capacity in order to cause the rejection of the call. In principle, the capacity check can be done in a tree-like fashion, where the nodes of the path are logically embedded as nodes of a virtual tree. Each node performs its local capacity check and then waits for all its children to report the result of their check. If any of these capacity check results is negative (an ABORT message is received from a child), the node sends an ABORT message to its parent on the tree. If all the results are positive (a COMMIT has been received from all children), a COMMIT is sent to the parent. Since nodes in our system can send direct message to other remote nodes, the tree structure is only logical and does not have to reflect any topological structure. By using, for example, a computation structure of a balanced binary tree, the capacity check can be accomplished in less than $2logn$ units of time (where $n$ is the length of the path). For relatively long calls, this provides a considerable improvement. In the following, we describe the details of this approach and the coordination of this process along the call setup path.

The process is initiated at each node after the reception of the SETUP message. We assume that the copied SETUP message contains both the forward route (partially stripped) from this NCU to the destination via all intermediate NCU's and the complete reverse route. The node can compute the length of the path $n$ from the reverse route. Upon reception of the SETUP message, all nodes can use an identical scheme for computing the capacity check tree. The local algorithm for computing a binary tree is as follows.

1. Number all nodes from the destination to the source from 0 to $n-1$.
2. Assign the nodes to groups of increasing sizes $(1, 2, 4, ...., 2^k)$ starting from the destination. The last group will contain fewer nodes if necessary.
3. For nodes of group $k \geq 1$ (with size less than or equal to $2^k$), assign parents in level $k-1$ such that no more than two nodes will share the same parent.

For example, suppose the path contains 13 nodes that are numbered 0-12. Node 0 is the root. Nodes 1 and 2 are assigned to group 1 and they report to 0. Group 2 contains 3, 4, 5, and 6. Nodes 3 and 4 report to 1, nodes 5 and 6 report to 2. Group 3 contains nodes 7, 8, 9, 10, 11, 12. Nodes 7 and 8 report to 3, nodes 9 and 10 report to 4, nodes 10 and 11 report to 5, and node 12 reports to 6. Since all nodes use the same scheme to compute the tree, there is no need for further coordination. The tree is then used to send the COMMIT as described.

## VI. Conclusions

In this paper, we have presented a distributed protocol for connection establishment and release in fast packet-switched networks. The protocol exploits the fast switching capabilities of the hardware to substantially reduce the possibility of contention in congested networks. This feature diminishes unnecessary resource reservation and, therefore, reduces blocking. We prove that the protocol ensures that a call is established if there are sufficient resources and that all resources are released after call termination or disconnection due to failures. An important feature of the protocol is that NCU failures do not affect existing calls and, upon recovery, the NCU can be easily reintegrated in the protocol.

## References


[1] J. S. Turner and L. F. Wyatt, "A packet network architecture for integrated services," in *Proc. GLOBECOM '83*, pp. 2.1.1 – 2.1.6.

[2] I. Cidon and I. S. Gopal, "Paris: An approach to integrated high-speed private networks," *Int. J. Digit. and Analog Cabled Syst.*, vol. 1, pp. 77–85, 1988.

[3] J. Y. Hui and E. Arthurs, "A broadband packet switch for integrated transport," *IEEE J. Select. Areas Commun.*, vol. SAC-5, no. 8, Oct. 1987.

[4] I. Cidon, I. Gopal, P. M. Gopal, R. Guerin, J. Janniello, and M. Kaplan, "The planet/orbit high speed network," Tech. Rep. RC 18270, T.J. Watson Res. Cent., Yorktown Heights, NY, Aug. 1992.

[5] J.-Y. Le Boudec, "The asynchronous transfer mode: A tutorial," *Comput. Netw. and ISDN Syst.*, p. 241, 1992.

[6] I. Cidon, I. Gopal, and S. Kutten, "New models and algorithms for future networks," in *ACM Symp. Princ. of Distrib. Comput.*, Aug. 1988, pp. 75–89.

[7] R. Cohen and A. Segall, "A distributed query protocol for high-speed networks," in *Proc. ICCC '88*,, Zurich, Oct. 1988.

[8] R. Cohen and A. Segall, "Distributed query algorithms for high-speed networks," in *Proc. Workshop High-Speed Netw.*,, Zurich, May 1989.

[9] A. Gopal, I. Gopal, and S. Kutten, "Broadcasting in fast networks," in *Proc. INFOCOM '90*, June 1990, pp. 338–347.

[10] A. Segall and J. M. Jaffe, "Route setup with local identifiers," *IEEE Trans. Commun.*, vol. COM-34, no. 1, Jan 1986.

[11] CCITT Blue Book, *Specifications of Signalling System No. 7, Volume VI, Fascicle VI.7*, 1988.

[12] R. Guerin, H. Ahmadi, and M. Naghshineh, " Equivalent capacity and its application to bandwidth allocation in high-speed networks," *IEEE J. Select. Areas Commun.*, vol. SAC-9, no. 7, pp. 968–981, Sept. 1991.

[13] H. Ahmadi, J. Chen, and R. Guerin, "Dynamic routing and call control in high-speed networks," *Proc. Workshop Syst. Eng. and Traffic Eng.*, June 1991, pp. 379–403.

[14] I. Cidon, I. Gopal, and R. Guerin, "Bandwidth management and congestion control in plaNET," *IEEE Commun. Mag.*, vol. 29, no. 10, pp. 54–63, Oct. 1991.

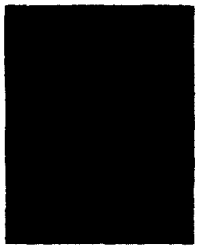
[15] A. E. Baratz and J. M. Jaffe, "Establishing virtual circuits in large computer networks," *Comput. Netw. and ISDN Syst.*, vol. 12, no. 1, pp. 27–38, Aug. 1986.

[16] J.M. McQuillan, I. Richer, and E.C. Rosen, "The new routing algorithm for the ARPANET," *IEEE Trans. Commun.*, vol. COM-28, pp. 5, pp. 711–719, May 1980.

[17] D. Bersekas and R. Gallager, *Data Networks*. Englewood Cliffs, NJ: Prentice Hall, 1992.

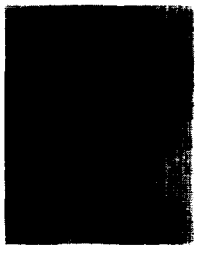
[18] A.E. Baratz and A. Segall, "Reliable link initialization procedures," *IEEE Trans. Commun.*, vol. 36, no. 2, pp. 144–152, Feb. 1988.

[19] I. Cidon, I. Gopal, G. Grover, and M. Sidi, "Real time packet switching: A performance analysis," *IEEE J. Select. Areas Commun.*, vol. 6, no. 9, Dec 1988.



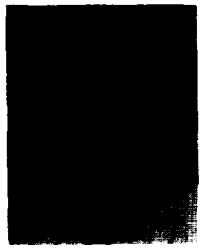
**Israel Cidon** (SM'90) received the B.Sc. (summa cum laude) and D.Sc. degrees in electrical engineering from the Technion–Israel Institute of Technology in 1980 and 1984, respectively.

From 1984 to 1985, he was with the Department of Electrical Engineering at the Technion. In 1985, he joined the IBM T.J. Watson Research Center, Yorktown Heights, NY, where he has been a Research Staff Member and a Manager of the Network Architectures and Algorithms Group involved in various broadband networking projects. Since 1990, he has also been with the Department of Electrical Engineering at the Technion. His research interests include high-speed local and wide-area networks and distributed algorithms. He currently serves as an Editor for the IEEE/ACM TRANSACTIONS ON NETWORKING. Previously, he served as the Editor for Network Algorithms for the IEEE TRANSACTIONS ON COMMUNICATIONS and as a Guest Editor for *Algorithmica*. In 1989 and 1993, he received the IBM Outstanding Innovation Award for his work on the PARIS high-speed network and topology update algorithms, respectively.



**Inder S. Gopal** (F'91) received the B.A. degree in engineering science from Oxford University, Oxford, England, in 1977 and the M.S. and Ph.D. degrees in electrical engineering from Columbia University, New York, NY, in 1978 and 1982, respectively.

Since 1982, he has been at the IBM T.J. Watson Research Center, Yorktown Heights, NY, serving in various technical and management positions. Currently, he is Manager of Broadband Networking in the High-Performance Computing and Communications Directorate. He is responsible for the development activities related to the TCP/IP routers that form the INTERNET backbone. He is also involved in the NSF/DARPA-sponsored AURORA Gigabit testbed and in several other research projects in the area of high-speed networking. His other research interests are in distributed algorithms, communication protocols, network security, high-speed packet switches, and multimedia communications. He has published extensively in these areas. He has received an Outstanding Innovation Award from IBM for his work on the PARIS high-speed network. He is currently an Editor for the *Journal of High-Speed Networking*. He has previously served as Guest Editor for *Algorithmica*, Guest Editor for the IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS, Editor for Network Protocols for the IEEE TRANSACTIONS ON COMMUNICATIONS, and Technical Editor for the *IEEE Communications Magazine*. He has served on several program committees for conferences and workshops.



**Adrian Segall** (F'88) was born in Rumania in 1944. He received the B.Sc. and M.Sc. degrees in electrical engineering from the Technion–Israel Institute of Technology, Haifa, Israel, in 1965 and 1971, respectively, and the Ph.D. degree in electrical engineering (with a minor in statistics) from Stanford University, Stanford, CA, in 1973.

After serving active duty in the Israel Defence Army, he joined the Scientific Department of Israel's Ministry of Defence in 1968. From 1973 to 1974, he was a Research Engineer at System Control Inc., Palo Alto, CA, and a Lecturer at Stanford University. From 1974 to 1976, he was an Assistant Professor of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology, Cambridge. From 1976 to 1987, he was on the faculty of the Department of Electrical Engineering at the Technion. He is presently Benjamin Professor of Computer Communication Networks in the Department of Computer Science, Technion-Israel Institute of Technology. From 1982 to 1984, he was on leave with the IBM T. J.Watson Research Center, Yorktown Heights, NY. His current research interest is in the area of high-speed communication networks. He has served as the Editor for Computer Communication Theory for the IEEE TRANSACTIONS ON COMMUNICATIONSand Editor for the *IEEE Information Theory Society Newsletter*. He was selected as an IEEE Delegate to the 1975 IEEE/USSR Information Theory Workshop, and is the recipient of the 1981 Miriam and Ray Klein Award for Outstanding Research and the 1990 Taub Award in Computer Science.