Simulation of packet and cell-based communication networks

Richard William Earnshaw

SIMULATION OF PACKET- AND CELL-BASED COMMUNICATION NETWORKS

Richard William Earnshaw,
B.Sc. (Dunelm)

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS OF THE COUNCIL OF THE UNIVERSITY OF DURHAM FOR THE DEGREE OF DOCTOR OF PHILOSOPHY (PH.D.).

APRIL 1992

Declaration

I hereby declare that this thesis is a record of work undertaken by myself, that it has not been the subject of any previous application for a degree, and that all sources of information have been duly acknowledged.

In the course of this research the following was included in an approved programme of advanced studies:

- A working visit to the Performance Engineering Division of British Telecom Research Laboratories, Martlesham Heath, Ipswich, August–September 1989.

R. W. Earnshaw, April 1992

Abstract

This thesis investigates, using simulation techniques, the practical aspects of implementing a novel mobility protocol on the emerging Broadband Integrated Services Digital Network standard.

The increasing expansion of telecommunications networks has meant that the demand for simulation has increased rapidly in recent years; but conventional simulators are slow and developments in the communications field are outstripping the ability of sequential uni-processor simulators. Newer techniques using distributed simulation on a multi-processor network are investigated in an attempt to make a cell-level simulation of a non-trivial B.-I.S.D.N. network feasible.

The current state of development of the Asynchronous Transfer Mode standard, which will be used to implement a B.-I.S.D.N., is reviewed and simulation studies of the Orwell Slotted Ring protocol were made in an attempt to devise a simpler model for use in the main simulator.

The mobility protocol, which uses a footprinting technique to simplify handoffs by distributing information about a connexion to surrounding base stations, was implemented on the simulator and found to be functional after a few 'special case' scenarios had been catered for.

Acknowledgements

I would like to acknowledge the invaluable help of my supervisor, Professor Philip Mars for his encouragement and suggestions during my work on this project; also to Steve Johnson, of British Telecom Research Labs (B.T.R.L.), who was my industrial supervisor. I am indebted to many people at B.T.R.L. for numerous detailed technical discussions on some of the finer points of performance engineering in telecommunications networks: notable amongst these are John Adams, who was involved in the early work on the Orwell protocol and who is a proposer of the mobility protocol discussed in chapter 6; and Garry Walley who spent several hours explaining the details of the Orwell protocol when both it and A.T.M. were still closed books to me. Thanks must also go to my colleagues and friends within the Telecommunication Networks and Music Technology research groups, whose humour and companionship have made my stay in Durham so enjoyable; also to the numerous technicians within the school who have kept the computers running. Finally, thanks must go to my parents and family, without whose love and encouragement this work would never have been possible.

To my parents

All the business of war, and indeed all the business of life, is to endeavour to find out what you don't know by what you do; that's what I called 'guessing what was at the other side of the hill'.

Duke of Wellington (1769–1852), Croker papers.

Contents

List of Figures

List of Plates

Chapter 1

Introduction

TELECOMMUNICATIONS HAS UNDERGONE ENORMOUS GROWTH during the last twenty years which correlates well with the development of the integrated circuit and progressively cheaper electronic systems. In 1970 communication networks in Britain were almost entirely used for ordinary telephony and the technology used was predominantly analogue; data communication was mainly conducted using telex machines, whilst in the U.S.A. new wide-area networks, such as ARPANET, were just being introduced. Today telephony networks, with the exception of the local distribution networks, are almost entirely digital; and high-speed data networks now span the world. Networks are being developed that integrate several services onto a single connexion, making more efficient use of the communications media. Data communication rates have increased by a factor of about a million in this time, whilst at the same time reliability has been improved. In the mid 1980's new services were introduced that enabled telephone conversations to be made and received without the need to remain in the same place; the scarce radio spectrum required for this was managed more efficiently by dividing the covered area into small cells, each with a different set of frequencies. The future also offers exciting developments: services that, even now, require significantly more bandwidth than those already integrated into a common network, will also be added to form a new Broadband Integrated Services Digital Network (B.-I.S.D.N.); it will be sufficient to provide bandwidths of several mega-bits per second to each customer and new services such as video telephony and audio-quality sound will be possible.

However, these new networking techniques still require a large amount of development and testing before they can be implemented; and a major tool in this process is simulation. Despite the increasing power of computers used for simulation they are not keeping pace with the increases in network speed (personal computers are about one hundred times faster than they were ten years ago, networks are between five- and ten-thousand times faster). To some extent this difference has been compensated for by the development of better modelling techniques and more appropriate computer languages that make expression of the problem more straightforward, but a large amount of 'brute-force' simulation is still needed when the system being studied is not yet fully understood. The falling cost of computer hardware now means that the differences between the development rates of the two technologies can be further reduced by *throwing* more hardware at the problem; this takes the form of getting several processors to co-operate in their simulation of a single task and, in the ideal situation, using k processors to simulate a network will produce the results in $1/k^{\text{th}}$ of the time. The von Neumann bottleneck of conventional computers is avoided by simply adding further processors each time more computing power is needed: however, in taking this approach the processing power problem is exchanged for a data interaction and synchronization problem.

Broadband communications is likely to form the next major growth area for the telecommunications industry and significant sums of money are being expended on the development of the new protocols and switching techniques that will be required. Many of the long established concepts that have been applied to conventional technologies (such as circuit-switching and packet-switching) no longer apply, or at best need careful re-examination. The high reliability of fibre-optic communications will still need to be pushed to the upper limits if the performance constraints that are being proposed are to be attained. The new broadband networks will mark an important move away from circuit switching with the introduction of the Asynchronous Transfer Mode (A.T.M.) as the paradigm for

broadband communications. Despite the similarities between A.T.M. and packet-switching, existing architectures cannot be used because the required capacities are orders of magnitude higher. New load control algorithms are being developed and one of these is included in the Orwell Slotted Ring protocol: it gives fair distribution of the ring's resource whilst still guaranteeing adequate delay bounds to permit the carrying of delay-sensitive services.

As already mentioned mobile communications is a relatively new service, but it is showing phenomenal growth characteristics; indeed the rate of growth is currently limited not by demand but by the rate at which the networks can be expanded. Further, the technology limitations of current implementations will shortly form a block to additional increases in network capacity: the cells covering an area are normally made smaller in areas of high demand, so that the frequencies in use for the radio link can be re-used more often; but there is a lower limit to the size of a cell that is determined by interference levels from neighbouring cells. Digital transmission techniques will help to overcome some of the interference problems; but then the size of the cells can be reduced to the point where the hand-off mechanism, which is used to pass a mobile from one cell to another, will be unable to cope with the increased load.

1.1 Outline of the Thesis

The main body of the thesis can be divided into three parts, which are covered in the following chapters. The first part covers simulation using distributed computing techniques: chapter 2 reviews some of the major techniques that have been used to ensure that the results obtained in this manner are not corrupted by the processing approach; while chapter 3 describes in detail a parallel simulator that was developed for the study of B.-I.S.D.N. networks and analyses its performance. The second part covers a data encapsulation and switching technique called Asynchronous Transfer Mode (A.T.M.), which is the transmission mechanism that will be used in a B.-I.S.D.N.: chapter 4 describes A.T.M. encapsulation

and switching techniques, while chapter 5 reviews a particular switch architecture, the Orwell ring, and presents simulation results from attempts to model the ring actions in a less computationally expensive manner. The final part, in chapter 6 covers a service that will have to be provided by a B.-I.S.D.N. network: the A.T.M. network simulator was used to test a novel mobility protocol that uses footprints to alleviate the signalling load presented to the network during hand-offs between cells.

Chapter 2

Simulation Techniques

THE DRAMATIC FALL IN THE COST OF COMPUTERS over the past ten years, coupled with the bottlenecks associated with single processor systems, has encouraged a new approach to computing techniques to be considered. Instead of concentrating all the financial resources of a computer in one, ultra-fast, central processing unit, multiple units can now be employed and made to work in parallel. New programming techniques, however, are required to exploit the concepts since a conventional programming language provides little scope for expressing parallelism (indeed, the opposite was encouraged in that potentially parallel tasks had to be expressed in a sequential manner).

One task that can benefit greatly from parallel-architecture computers is simulation: these often model many tasks that would operate in parallel in the real system so there is a natural parallelism that can be exploited. Unfortunately, the approach most commonly adopted, discrete-event simulation, is a good example of how a programming methodology has been developed to suppress expression of the parallelism. The whole process is reduced to a series of events that occur one after another; each event is assumed to occur at a definite point in time, and the effects are transitive; multiple events that occur at the same time and interfere with each other are not permitted. This makes discrete-event simulation highly applicable to telecommunication networks where the system operates in a sequential-state mode (a connexion is established, a message is sent etc.); but is a poor approach when time-continuous systems (such as fluid flow) are being studied.

There have been several approaches proposed for implementing discrete-event simulation concepts on multi-processor architectures so that the parallelism of the problem can be exploited. By far the most common approach is to split the objects of a simulation that would in reality run in parallel into separate tasks and to simulate each of these in parallel. An orthogonal approach, however, is to split up the processes involved in simulating the entire problem, such as generating random numbers or manipulating the event set, and to run each of these on a separate processor. It is commonly found in simulation work, particularly of telecommunication networks, that the same problem must be simulated many times in order to gain an insight into the statistical stability of the system; so another approach that can be adopted is to run copies of the same simulation with different random-number sequences on separate processes: each result will take just as long to extract as before, but the degree of confidence in it should be available immediately instead of several runs later.

This chapter reviews techniques that have been used in structuring the software used for simulation on parallel machines: it concentrates most upon the technique of distributing the simulation objects amongst several processes and the various techniques that can be used to do this whilst maintaining the integrity of the results. The final section considers how the event set for a simulation is manipulated and how this is affected by distributing the simulation objects.

2.1 Distributed Simulation Objects

The aim when parallelizing a simulation using distributed simulation objects is to extract from the system being simulated those processes that run concurrently in the real system, and to simulate these processes in parallel. A telecommunications network is an ideal example of a system that can be parallelized in such a manner: each node in the real network operates independently and in parallel with all of the other nodes. Hence, if each of the nodes in the real network can be simulated

on a separate processor then significant speed-ups may be possible, particularly for large networks.

However, by simulating a system in such a manner a processing power problem is exchanged for a synchronization problem. In a conventional, serial, simulator the pending-event set is maintained as a single entity: the simulation consists of picking from the pending-event set the event with the earliest time, simulating it, and then repeating this process until some termination condition becomes true. As a result of simulating an event more events may be generated which are stored in the pending-event set; since the new events are always generated for some interval of time after the one currently in progress, the net effect is that the simulation as a whole moves monotonically forward in time towards its completion. When the simulation is distributed ideal speed-up is obtained when each of the processes in the simulation is able to execute the event at the head of its event list. Unfortunately, simply processing each of the lists entirely independently can lead to *causality errors.*

In a discrete-event simulator changes in the state of the system occur at discrete points in time (hence the name); when there are no changes in state there are no events in the pending-event set and, hence, the simulator is very efficient in that it only considers the times of interest, skipping over the idle periods. For a distributed simulation it is possible for one process to have a series of events with large jumps in time, while a neighbouring process might have a set of events very closely spaced in time; it is then also possible for the one with events with small time increments to issue a message to the other process and for this message to be scheduled for execution at a time prior to the currently active event's: if earlier receipt of this message would have altered the way in which the current, or previous events should have been processed then a *causality error* is said to have occurred; in such cases the fidelity of the simulation is compromised and the results must be regarded as suspect. Clearly such a scenario is unacceptable in the general case and some of the potential speed-up needs to be sacrificed in order to guarantee the validity of the results. There has been much research in

this field and generally the results can be divided into two categories known as conservative and optimistic synchronization.

2.1.1 Conservative Synchronization Techniques

The initial work on conservative synchronization methods was performed by Chandy, Misra and Holmes [1, 2] and simultaneously by Bryant [3]. Chandy's method uses the Communicating Sequential Processes (C.S.P.) approach developed by Hoare [4], enabling the amount of memory required to be tightly bounded. The simulation model consists of three basic types of object: queues, which have exactly one input and one output and pass messages with some delay function; forks, which have a single input and two or more outputs; and merges, which have two or more inputs and exactly one output. In addition there are sources, which generate messages, and sinks, which absorb them: both of these are simply special cases of the queue object. The simulation can then be constructed using these objects as building blocks. During the simulation each object has three phases: a link accessibility phase, during which the links eligible for communication are determined; a simulation phase in which messages are processed and new messages are prepared for the links eligible for communication; and a message exchange phase during which period the object forwards messages over the output links that were enabled during the first stage, and receives messages from those input links that were enabled. Both transmission and reception must operate concurrently to avoid deadlock — all the links determined as eligible during the first phase must complete a communication before the next iteration can begin. Time within the simulator is fully distributed in that there is no global clock and there is no single process that directs an object to process a message: therefore, to avoid causality errors, a merge object can only select a message for processing when each of its input links has a message available; each of the messages is marked with a time by the transmitting object and hence the merge process simply selects the message with the smallest time-value for processing. If the simulation model is totally loop free, then it is a trivial matter to show

that the simulation will run to completion, provided that a special termination message is generated when the simulation completes; this is propagated through the network to flush out the remaining messages. However, if the network of objects contains a fork that is followed by a merge (possibly with intermediate objects), for example figure 1, then such a simulation can deadlock due to the

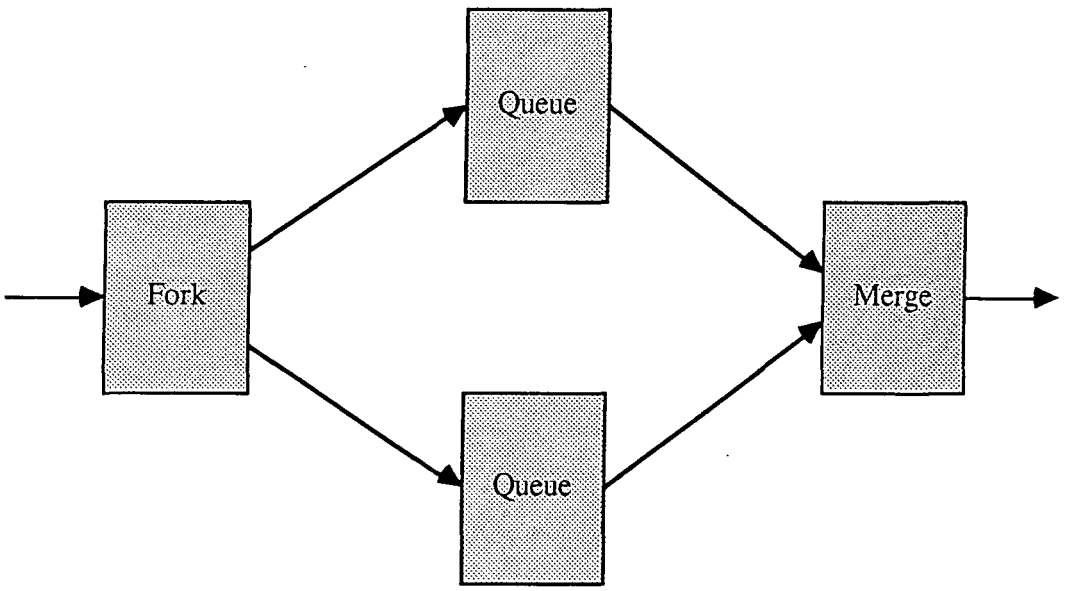

Figure 1: A feed-forward network of processes that can deadlock during conservative simulation if NULL messages are not used.

synchronizing approach provided by the C.S.P. model: this can occur if a series of messages are sent along one of the paths but another path remains idle: the merge object cannot process the first of the messages until it receives a message on all of the other links and consequently cannot receive a second message on a link for which it already has a message waiting; the fork object cannot send a message along the idle link until it has finished sending messages along the busy link. To avoid deadlock in such circumstances each output link that is determined during the first phase to be eligible for a communication, but has no real message, transmits a NULL message.

As already noted, each message that is transmitted between two objects is associated with a simulation time, which is the time at which the entity that the

message represents would have moved from one object to the other in the physical system: the message and its time are commonly represented by the tuple (t, m). A NULL message, the tuple (t, NULL), does not represent any entity moving between processes and hence has no counterpart in the physical system: it is used simply to inform another simulation object of the time elsewhere in the simulation; this can enable the merge process to carry on accepting messages on a link safe in the knowledge that no new message will arrive on that link with a time-stamp earlier than the last NULL message received. One of the main drawbacks of the the NULL-message approach is the number of such messages that can be in the system relative to useful messages; it is a particular problem when simulating low loads or when feedback loops exist within the simulation (indeed, Chandy and Misra do not mention simulations of such a nature in their original work). Misra [5] has established several conditions under which NULL messages can be annihilated: these include, for example, a second message arriving on a link before a prior NULL message has been processed; in this case the NULL message can be safely ignored since the timing information it carried has now been outdated by the new message. Other techniques involve delaying, in real time, the transmission of the NULL message in the hope that the above will occur more often.

Significant improvement in the performance of a distributed system can be achieved if, instead of holding a message in an object until the clock times on all of the inputs reach the time of departure, the message is processed and passed on to the next object as soon as the causality of the simulation can be ensured. In doing this the simulation objects loose the information of the queue length and additional calculations are required if the length of the simulated queue needs to be known [1]. It should be noted, however, that such a technique cannot be used with priority-queueing structures or pre-emptive messages since the message processing order cannot be determined in advance. The ability to pipeline and pre-calculate the output messages in this manner was formalized as the look-ahead capability of the simulation by Fujimoto [6, 7]. The look-ahead of the

simulation is defined as the interval of time during which the output events can be determined solely from information already received. If an event arrives at a time t_{cause} and generates an output event at time t_{effect} then the look-ahead ratio (L.A.R.), is defined as:

$$L.A.R. = \frac{t_{\text{effect}} - t_{\text{cause}}}{look\text{-}ahead}. \quad (1)$$

A physical system that is well suited to simulation in this manner will have a low L.A.R. value, since this corresponds to a high degree of look-ahead. Work by Reed, Malony and McCredie [8] and by Reed and Malony [9] has suggested that central server queueing systems were unsuited to this type of simulation algorithm, but Fujimoto has since shown that simulations of this form are possible provided that the look-ahead is exploited effectively [6].

Nicol [10] showed that it is sometimes possible to improve the look-ahead of a simulation (and, therefore the indications provided by the NULL messages) by calculating a futures list: this is a list of the job processing times of future jobs that have yet to arrive at this node. For the Chandy-Misra algorithm to work, each logical process must have a minimum service time, ϵ, so that NULL messages flowing round a loop will increase in time and move the simulation forward; in practice, however, since a large number of statistical distributions have no minimum processing time, ϵ has to be quite small and this can lead to very poor performance: the futures list can enable a node with no jobs pending to predict the *minimum* time at which a job would leave the node if it were to arrive immediately. Nicol has implemented the Futures List scheme, along with an appointments protocol, on a shared-memory, multi-processor system and reports that the speed-up can often approach the ideal, even at relatively low loads, provided that the connexion topology of the simulated network is not 'too rich'.

A significant reduction in the population of NULL messages has been achieved by Cai and Turner [11]. By adding to some of the NULL messages information about the route that the message has followed and details of look-ahead elsewhere in the network, real messages can be accepted earlier by merge tasks; fewer NULL

messages need to be generated since the minimum increment of ϵ can be improved upon. Su and Seitz [12] have used several variations of the basic Chandy-Misra scheme ranging from Eager Message Sending (the original technique) to Demand-driven (where the messages are only transmitted at the request of the receiver for more information) and include self-adaptive variants in between. It was found that, while it was usually possible to find an algorithm that was better than the basic scheme, no single algorithm was consistently better for all simulation configurations. De Vries [13], however, uses decomposition techniques when partitioning the simulation elements to assist in predicting message arrival times when an input link has no message; in this way the only time a simulation element must block is when the link with the lowest input time has no message: this leads to fewer NULL messages at the expense of adding more knowledge of the simulated environment to the synchronization model.

Chandy and Sherman [14] have proposed the use of conditional events in place of NULL messages for advancing the simulation. In a sequential simulation a conditional event is defined as any event that will be executed next, provided that there are no events scheduled for earlier execution; this means that for a sequential simulator there is one definite-event and many conditional-events. In a distributed simulation, each logical process does not necessarily have a definite event at the head of its list since messages may still arrive from other processes that will pre-empt them. However, there will be at least one, and probably more, definite events somewhere in the simulation; it is the task of the simulation controller to promote as many conditional-events as possible by sharing timing information between the tasks. Good speed-ups have been obtained for scenarios that usually showed poor results using conservative algorithms.

Several researchers have developed environments for producing general-purpose conservative simulators. These include the Object Library for Parallel Simulation (O.L.P.S.) by Abrams [15] which is a set of C++ classes; a development on top of this library is the Common Programming Structure (C.P.S.), again by Abrams [16] which is a programming structure, as opposed to a language, that

enables the simulation to be structured in a manner that permits the simulation to be run under either a conservative environment or an optimistic one. The approach adopted by Nicol [17] was called YAWNS (Yet Another Windowing Network Simulator) and was an attempt to allow several distributed simulation techniques to be compared experimentally without the need to recode large sections of the model under simulation. A third technique, called YADDES (Yet Another Distributed Discrete Event Simulator) [18, 19], uses a compiler pre-processor in a similar manner to the well known yacc language compiler to permit the simulation hooks to be added to the detailed parts of the simulated model in an implementation independent manner; the resulting YADDES script can then be compiled and linked for either sequential, conservative or optimistic simulation.

2.1.2 Optimistic Synchronization Techniques

The essence of a conservative algorithm is that the simulation proceeds only when the integrity of the results can be guaranteed: conversely, an optimistic algorithm always proceeds, but saves the state of the simulation regularly so that, if an error is detected, the results can be *rolled back* to a known state and then re-simulated taking into account the new information. The advantage of this approach is that, in the absence of messages from other processes, the simulation can proceed without large numbers of NULL messages or the resynchronizations that they imply; this can mean that performance is substantially better than conservative techniques when the look-ahead is poor. The penalty is the overhead of storing copies of the simulation state at regular intervals and restoring one of these when an error is detected: in addition to *rolling-back* the local state any messages that were transmitted during the period being re-simulated may have to be cancelled; this is done using *anti-messages* which annihilate the corresponding original message.

The initial work on optimistic algorithms was performed by Jefferson [20] which led to the first version of the Time Warp simulation environment. This was implemented in GLisp (a dialect of Lisp) on a network of Xerox SIP-1100

processors [21]; more recent implementations, known as the Time Warp Operating System (TWOS), have been written in C and run on various hardware architectures but notably the Caltech Hypercube [22].

Optimistic algorithms are commonly said to be running using Virtual Time (after Jefferson's original paper [20]). This is an analogy between the method by which they manage time and the paging method of Virtual Memory computing architectures: with virtual memory the processor is gambling, when a page of memory is switched to disk, that the page is not likely to be referenced again before those that have been kept; with virtual time the gamble is that by simulating an event before the causality effects can be guaranteed, no causality errors will in fact occur: the penalty in both cases is that time is lost when the gamble does not pay off.

The implementation of virtual-time algorithms requires not only that the list of future events needs to be stored, but that copious amounts of information about earlier states that the simulation has passed through must also be kept against the eventuality of a roll-back. Such information includes copies of the state tables taken at periodic intervals during the simulation (these are the check-points — when a roll-back is required the state is restored to the last check-point earlier than the time required and simulation restarted from there); the list of events to be processed, which will usually be the messages received from other processors; and copies of the messages sent to other processors, to enable anti-messages to be created if required. With simulations consisting of thousands (perhaps millions) of events then clearly it would be impossible to continue to keep all of this information throughout the entire run, since it would rapidly consume the entire amount of memory on a processor. As the simulation progresses each process's Local Virtual Time (L.V.T.) moves forward and backwards and a time known as Global Virtual Time (G.V.T.) can be calculated based on the minimum of all of these across the entire network (slight complications arise in the calculation due to the delays that messages incur whilst traversing the network). Any saved information about the past state of the simulation that is earlier than the last state

saved before G.V.T. can be safely destroyed since a process can never roll back to an earlier time; the process of releasing memory in this way is known as *fossil collection.* Results of the simulation earlier than G.V.T. are said to have been committed; in particular, any messages destined for output to a device outside of the virtual-time mechanism (for example, a disk or a terminal) can only be safely written once the time on the message is less than G.V.T..

The single largest overhead of this technique is the saving of the simulation state at regular intervals throughout the simulation; this involves making a copy of potentially large amounts of state information that may or may not have changed since the state was last saved. In an attempt to reduce this overhead dedicated hardware has been designed to perform this operation in a near transparent manner [23, 24]. The Roll-back Chip (R.B.C.) maps frames of memory into the same address space with each frame representing a saved state. Each time the state is saved a new frame is mapped in, but the state information is not copied across, thereby saving valuable bus bandwidth. Instead a tagging technique is used that marks a memory location in a particular frame as containing valid data: a write to the state memory is always made to the current frame, causing the valid-flag to be set; a read consists of selecting the most recent frame with the valid-flag set and returning its contents. Care must now be taken when undergoing fossil collection: state information that has remained unchanged for a long period must not be destroyed but copied into one of the remaining valid frames. Other limitations lie in the finite number of frames that can be managed by the R.B.C. (16 in the version described in the literature), problems associated with the interaction with a cache (particularly if the cache is of the delayed-write type) and its use in a virtual-memory management system; all of these can be overcome with care.

As mentioned earlier the calculation of Global Virtual Time plays a vital rôle in the commitment of results and the reclamation of memory from previously saved states. Unfortunately this calculation requires some indication of the state of the entire simulation (the virtual time at each processor) and is not amenable to a truly distributed calculation. In practice a single processor (usually labelled

zero) is responsible for establishing new estimates of G.V.T. and for informing the other processors of these. The calculation is based on the minimum of each node's virtual time and the message receipt time (in practice, since flow control in the simulator is commonly implemented by returning a message to its originator, the message transmission time is used in place of the receipt time); using conventional G.V.T. calculations it takes, typically, 5 message acceptances and 3 to 5 message transmissions per processor taking $O(N)$ time using a broadcast algorithm supported by the simulation hardware. Bellenot has shown [25] that it is possible to reduce this to less than 4 message transmissions and, without using a hardware broadcast, to reduce the time taken to $O(\log N)$.

The overall aim of distributed simulators is to minimize the total simulation time by extracting the maximum amount of concurrency from the model. In an optimistic simulation a processor proceeds whenever it has tasks to process, based on the assumption that if it were blocked it could never do useful work but if it were to continue with 'speculative' work this might also be useful. However, this can lead to a substantial amount of work that needs to be undone when it is found to be incorrect. Reiher *et al.* [26] suggested that it might be possible to improve the performance of virtual-time mechanisms (such as Time Warp) if the optimism were limited in such a way that tasks doing work that was likely to be undone at a later date were discouraged in favour of tasks that were more likely to be doing useful speculative work. The two methods tested were a windowing mechanism which throttled back processes whose local virtual time was far into the future and a penalty technique that penalized process that did poor speculative work (i.e. that was subsequently rolled back) in favour of processes that had done useful work. Neither technique was found to produce significantly better results and it was found that performance gains were unpredicatable.

Several of the simulation environments mentioned previously in conjunction with conservative simulation also have alternative libraries that can be linked in to make the simulation run in an optimistic manner: notable among these are YADDES [18, 19] and the O.L.P.S./C.P.S. systems [15, 16]. However the two

most significant implementations are TWOS (a U.S. D.o.D. funded project and as such with restricted access) and the commercially available Sim++ system by Jade Simulations [27].

2.1.3 Hybrid Techniques

The techniques of conservative and optimistic simulation described so far represent merely the extremes of a whole myriad of different approaches; in practice it is possible to 'borrow' aspects of one of the techniques for use in a methodology that lies predominantly towards the other extreme. Reynolds [28] categorized these as a whole spectrum of different options and classified several of the approaches already taken: he also identified several other approaches that, as yet, remain unexplored.

A typical example of a hybrid technique is Gimarc's Hierarchical Roll-back [29]. The simulation is structured as a tree such that each process can only communicate with its parent and any children. Within each node the process proceeds in an optimistic manner and can roll back as required; however, non-local roll-backs are restricted to sub-trees and the sub-trees themselves interact in an essentially conservative manner. Time messages (effectively NULL messages) propagate up and down the tree and provide for overall advancement in simulation time and for fossil collection. Another example is Lubachevsky's Filtered Roll-back approach [30] which applies his own Bounded Lag algorithm [31] on top of an optimistic simulator in an attempt to contain the amount of roll-back that may be required during a simulation run. (He presents an example where the amount of time spent rolling back increases with each step forward in the simulation when a purely optimistic algorithm is used.)

The overall aim, of course, with all the techniques presented here is that the simulation itself should run as quickly as possible. Lipton and Mizell [32] present calculations that show that under some simplifying assumptions, Time Warp can out-perform the Chandy-Misra algorithm by a factor p for a p-process model, but that the reverse does not hold: the Chandy-Misra model can never outperform

Time Warp by more than a constant factor. In an attempt to establish the optimum performance that can be obtained from a parallel simulator, Swope and Fujimoto [33] have developed a tool that uses an oracle to establish whether or not it is safe to proceed. The system works by making two passes through the simulation: during the first pass, which is performed using some traditional method the causality information is loaded into the oracle; during the second pass the oracle is used to control each local process so that it runs the optimal sequence of computations and only blocks when it *knows* that a causality error will occur. The time taken for the second pass can be used as the optimal value for running the simulation in a distributed manner.

2.2 Concurrent Simulation

One of the main alternatives to distributed simulation is a technique developed by Jones [34, 35, 36]. In this method the processors co-operate in manipulating a single event list; for this reason the technique is best suited to shared memory multi-processing systems. A discrete-event simulation has three distinct phases associated with the simulation of each event: extracting the next event from the pending-event set; manipulating the state space of the simulation; and scheduling future events. If a causality constraint similar to the look-ahead in distributed simulation is applied to events in the pending-event set, then the causality relationship between events scheduled during some time interval, ϵ, from the time of the head event is defined. A multi-processor system may therefore co-operate in the processing of the event set: the event selection and manipulation of the state space must be performed as a single atomic action (to preserve causality and data integrity), but several processors may be adding new events to the event set while one processor is manipulating the state space.

The algorithm can be improved upon if the state space is subdivided and a processor is able to 'lock' access to those values it is referencing: each process can then proceed with the algorithm in figure 2.

repeat
lock event set
extract event
lock areas of state space
unlock event set
manipulate state space
unlock state space
schedule new events
until *(termination condition is reached)*

Figure 2: Jones' algorithm for concurrent simulation.

Provided two (or more) consecutive events in the set require mutually exclusive areas in the state space then concurrent simulation may be performed in this region as well. (Note that area in the state space must be locked even if the value is only being read since otherwise an event scheduled for later execution may change the value prematurely and cause a causality error.)

2.3 Other Parallelizing Techniques

So far, only a couple of decomposition methods have been discussed, but there exist several other techniques, each with its advantages and proponents.

The advent of local- and wide-area networks has, in general led computing systems away from centralized machines to a more distributed network of slightly less powerful, but significantly less expensive, processing elements such as workstations and personal computers. Such networks of processors can be used as a simulation system and many of the decomposition methods already mentioned can be used to synchronize the simulation: but other factors also apply, notably that different processors in the network have different characteristics that make them best suited to different aspects of the simulation. In addition, the bandwidth available for inter-processor communication is generally lower and the message delivery time is often significantly higher (particularly if guaranteed delivery is

required — the usual assumptions of error-free delivery is only available at the expense of a substantial protocol overhead). Techniques such as dead reckoning (predicting the future behaviour on past experience) are often used to reduce the synchronization overhead. Such simulation techniques are often termed Heterogenous Distributed Simulation [37].

2.3.1 Replicated Experiments

Simulation is commonly used for the study of statistical or stochastic systems; it is therefore necessary that either very long runs, or many shorter runs, of a single experiment be performed in order to establish the degree of confidence in the results obtained. This has led to the proposal that replicated experiments be used: instead of several processors contributing towards a single simulation run, each processor runs an independent copy of the simulation with its own set of random number seeds. The potential advantage of this approach is that there is no synchronization required between the processors, and variance reduction techniques can be used to tighten the bounds on the results [38, 39]. Against these, it should be noted that significantly greater computing resource may be required at each node than when all the processors are working on parts of the same task: an extreme example of such a processing network is the transputer tree developed in Durham [40, 41, 42] which has about 160 processing elements, but only four kilo-bytes of memory for each processor.

2.4 Event List Management

The central feature of a discrete-event simulation is the pending-event set: this is the set of known tasks that must be performed at various times in the future. The set is dynamic in that events are removed from it as they are simulated and further events are added to it as a consequence of performing the simulation. Each event in the set requires three parameters: the time at which the event is to be executed; the function to be performed; and a data-set to manipulate,

which may be empty for some functions that manipulate the global state of the simulation. The principle behind the manipulation of the event set is that the simulation moves monotonically forward in time by selecting the event that is the least distance into the future: the result of simulating each event will be to generate zero or more subsequent events for execution at some later time (clearly it is impossible to generate an event that should have been executed before the current one).

To facilitate rapid selection of the next event from the pending-event set, the 'obvious' approach is to maintain a sorted list: this means that selecting the next event is trivial since it is at the head of the list (i.e. it takes O(1) time), while inserting an event will typically involve searching through half the list to find the correct position (this assumes that events in the pending event set are evenly distributed) hence the insertion takes $O(n)$ time for a pending-event set of size n.

2.4.1 Improved Pending-event Set Manipulation

For large simulations n can become large (perhaps a few thousand events) and hence the time spent searching the event list to insert new events can form a significant proportion of the simulation time, particularly if the number of manipulations of the simulation state is small for each event simulated. There has, therefore, been significant effort expended on attempts to manage the pending-event set in a more efficient manner: a review of several of the most common techniques is given by Jones [43]. Jones presents results showing that for very small sets (< 10) a linear list is optimal, that for medium sized sets (< 200) the Two List [44] procedure performs very well (this gives $O(\sqrt{n})$ search time), while for large sets the Splay Tree method [45] gives probably the best performance ($O(\log n)$ search time).

In the two-list method the pending-event set is divided into a small set of sorted events and a larger set of unsorted events, the sorted set containing those events that will be simulated in the near future. The intention is that the majority of events will normally be placed in the larger, unsorted, set requiring O(1) time

for a store; however, the sorted list will occasionally become empty, whence a new threshold time is calculated and all those events in the unsorted set with time less than the threshold are transferred to the sorted list. Blackstone *et al.* [44] show that the optimum size for the sorted set is $\sqrt{n}$, leading to $O(\sqrt{n})$ searching time on average.

The splay-tree method, as with most attempts to store the event set so that search times are of order better than $O(\sqrt{n})$, organizes the event set in the form of a tree structure (a balanced tree giving a search time of $O(\log n)$). However, a static tree-structure would rapidly become very poorly balanced since insertions would primarily take place on one side of the tree, whilst deletions would be from the other side: to overcome this problem the tree needs to be re-organized as each insertion or deletion is made to ensure that the balance is kept. Producing a fully balanced tree after each change would clearly require far too much time, and is, in fact, unnecessary; instead a re-balancing heuristic, known as splaying, is used that has the effect of bringing the referenced item in the tree to the root and, in the process, roughly halving the depth of each of the nodes in the path between the root and the referenced node. It has been shown [45] that the amortized search time[1] using this heuristic has a bound of $O(\log n)$.

2.4.2 Distributing the Pending-event Set

When the simulation is distributed between several processes using distributed simulation objects each process maintains its own event set containing those events that that processor must execute (in the pure Chandy-Misra form this is the list of input channels and their lower time bounds). Jones [34], when arguing in favour of concurrent simulation states that partitioning the total event set in this manner leads to inefficiency. The total size of the event set, n, when distributed over m processes would be n/m events per process, leading (with an optimal searching algorithm) to a speed-up, S, of

[1] *Amortized time* is the time per operation averaged over a series of worst-case operations.

$$S = \frac{\log n}{\log \frac{n}{m}}. \tag{2}$$

Solving this for $S = 2$, he claims that to double the speed of the event-set processing would require $m = \sqrt{n}$ processors. This speed-up, however, is the *per transaction* speed-up, and fails to take into account the fact that the number of references by each process to its local event list in the distributed case is further reduced by a factor of m. If there are k events simulated during the simulation, then on average each processor in the distributed version will only process k/m of them. This leads to a speed-up of

$$S = \frac{k \log n}{\frac{k}{m} \log \frac{n}{m}}, \tag{3}$$

i.e.

$$S = \frac{m \log n}{\log \frac{n}{m}}; \tag{4}$$

hence the speed-up is *better* than simply linear. It must be noted, however, that the speed-up of the entire simulation is unlikely to approach this bound since the time spent simulating each individual event is unlikely to be reduced as a result of distributing the simulation and, in addition, time spent resynchronizing the processors must also be taken into account: it can, however, be taken as a reasonable upper bound on the potential speed-up that could be obtained. Situations such as this are discussed by Helmbold and McDowell [46].

Perhaps the validity of equation 4 requires some notional justification; this can be seen by considering the manner in which the performance improvements were obtained for the conventional system. All of the methods used for this involve partitioning the event set so that fewer accesses were required to store or retrieve an event; partitioning the pending-event set between several processes clearly takes this partitioning process one stage further — this leads to the $\log(n)/\log(n/m)$ term in the speed-up. In addition each processor only handles $1/m^{\text{th}}$ of the total number of events simulated, leading to the factor of m in the speed-up. Partitioning the pending-event set in this manner for a conventional simulator would

be detrimental since the event scheduler would have to check each of the sets before it would be able to select the event to simulate; this would take $O(m)$ time, more than cancelling out the other benefits. The result shown in equation 4 is based on the use of a splay-tree event-storage scheme; for the two-list scheme the speed-up would be $m\sqrt{m}$; and for a linear-list scheme it would be m^2. Further, since the size of the sets is reduced by the partitioning the simulation, it might be beneficial to consider using a less complex storage scheme which has a lower per-iteration overhead.

Chapter 3

A Distributed Simulator for a Transputer Network

TO ISOLATE THE SIMULATION MODEL, as far as possible, from the implementation details of the hardware, the simulator was structured in a hierarchical manner; each layer builds on the abstraction of the layer below in a similar approach to that of the I.S.O. seven-layer model. At the lowest layer lie the transputer processors in a dynamically reconfigurable array. On top of this a multiplexor task[1] on each processor provides the abstraction of virtual channels between each task in the simulation, regardless of where the tasks are mapped in the processor network. A simple packetizer layer hides the fact that the channels in the multiplexor (and, indeed, the physical channels of the transputer itself) work most efficiently when presented with large packets as opposed to a series of very small ones. A synchronization layer uses the packet-layer processes; it ensures that each message is correctly marked with a time-stamp on dispatch; and uses this at the receiver to maintain synchronization: the layer is optional; if there is no definable synchronization between two tasks (for example, diagnostic messages destined for the console) then the channel can be declared asynchronous and the packet layer accessed direct. Finally, in parallel with the simulation model and the synchronization layer, an event manager is responsible for scheduling components of the simulation model in the correct sequence. The overall hierarchy is summarized in figure 3.

[1]The terms 'task' and 'process' are used interchangeably: a process runs on a processor; but to avoid ambiguity in such circumstances, 'task' is normally used.

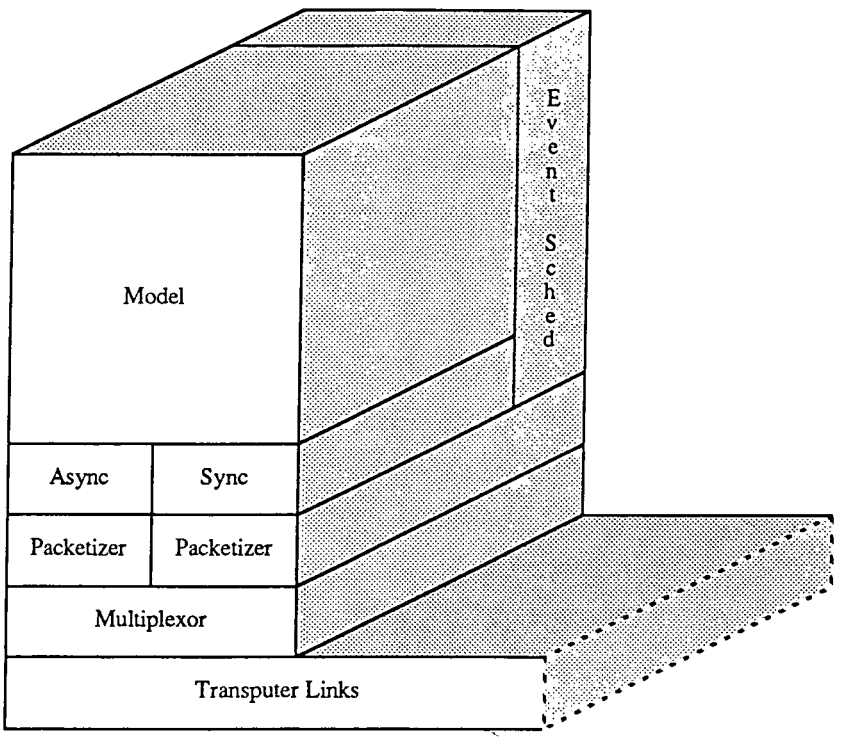

Figure 3: The overall hierarchy of the simulation model. The Event scheduler is a control-plane for the upper layers.

Given an array of unbooted processors, the non-trivial problem of how to configure the simulation is then considered, along with how the information necessary for this can be obtained from the user. Finally, the performance of the simulator, known as the A.T.M. Network Simulator, is briefly discussed in terms of its efficiency and the degree of concurrency achieved.

3.1 The Transputer Network

The transputer network used for this project was originally designed for use as a high speed circuit-switched network simulator, with code written in occam; subsequently, a traditional packet-switched network simulator was also developed using the same language [47, 48]. The processor network consists of up to 31 simulation, or worker, transputers, each with up to 16Mbytes of memory (the current implementation consists of 13 such processors each with 1Mbyte of memory);

the original T414 transputers were replaced with T800 processors approximately half-way through this project, leading to the speed of the simulator being more than doubled. The transputers are connected with a double layer of C004 link switches which enables any link on each of the worker processors to be connected to a link on any of the other processors; this flexibility enables the network to be configured to almost any topology (including those without linear chains through the processors, as some other hardware configurations enforce) so that the system being simulated maps closely onto the processor network, and enables the path length required when passing messages between processors to be kept to a minimum. Finally, a layer of control processors are used to connect between the host transputer (inside an I.B.M. P.C./A.T. compatible) and the link switches; one is connected to the link-switch programming interface, while both can be connected, via the switches, to any of the worker transputers. An optional transputer-based graphics card can also be connected at this layer. Figure 4 shows a functional representation of the hardware used.

3.2 The Multiplexor

The multiplexor is the lowest layer of the simulator kernel; it is responsible for the delivery of messages from one task in the simulator to another, regardless of the topological mapping of either the tasks or the processors upon which they are running.

Each transputer in the network is allocated exactly one multiplexor task; all other tasks desiring to communicate with tasks on another processor do so by communicating indirectly via the multiplexor (figure 5). If two tasks that communicate are on the same processor then it is, of course, possible for them to be directly connected. The result is that for large simulations a simulation task may have many of its channels connected to the multiplexor; the routeing decisions that the multiplexor makes are based solely upon the channel from which each message is received.

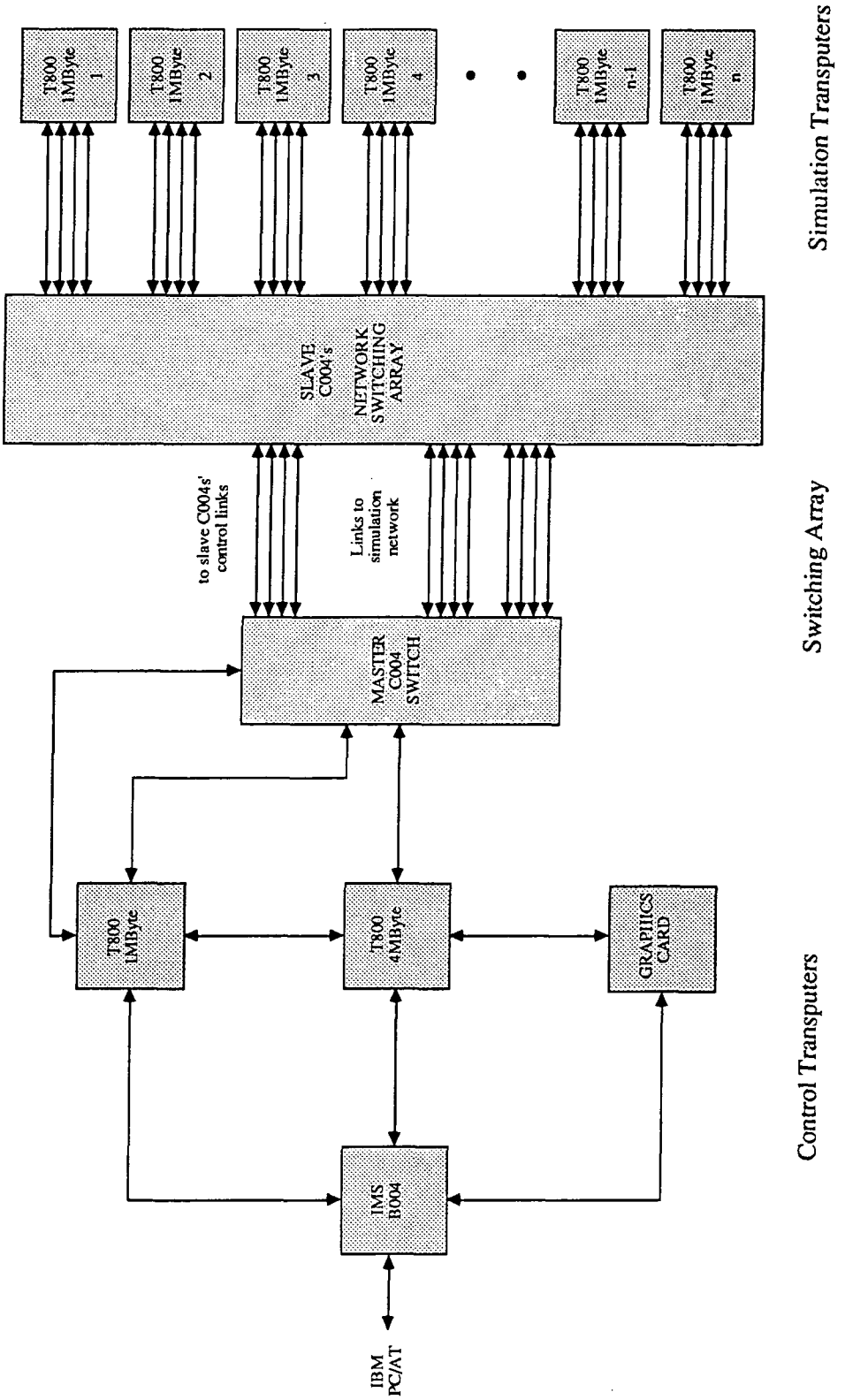

Figure 4: High-speed transputer-based network simulator — Hardware configuration.

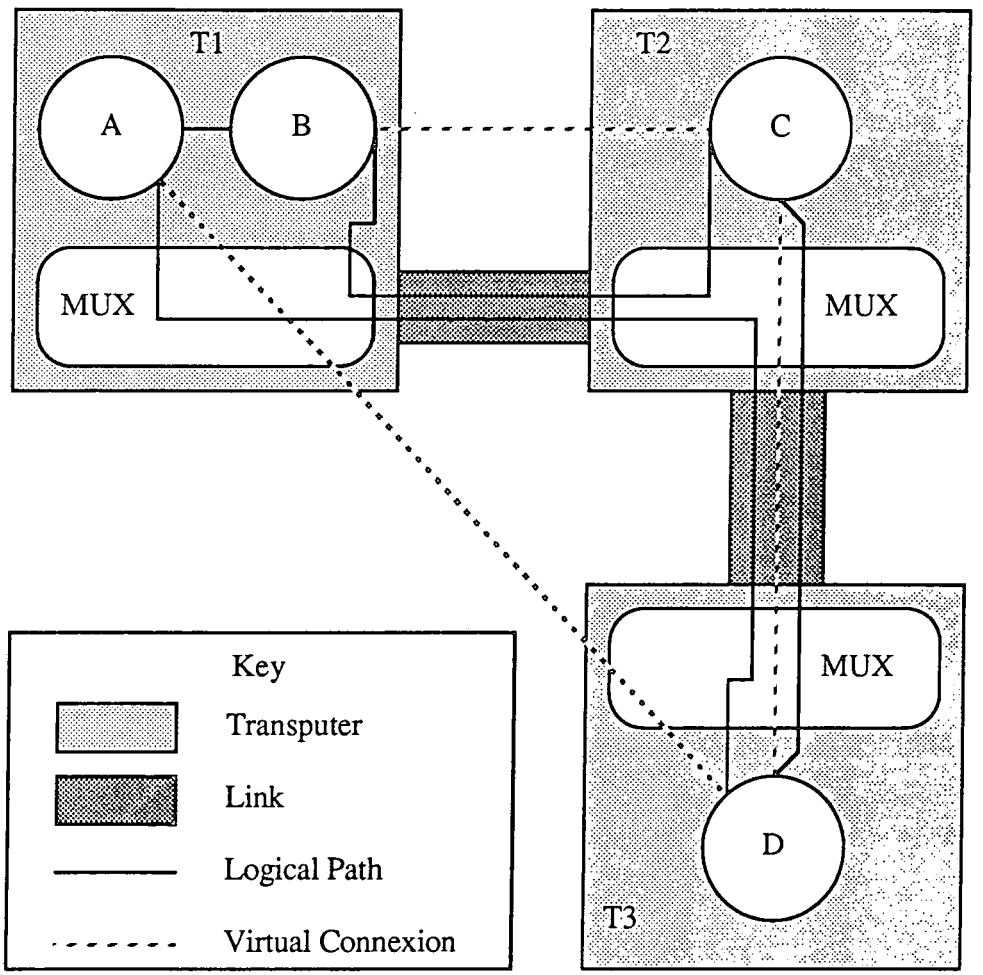

Figure 5: Multiplexor processes run on each node to provide virtual links between each task in the simulator.

3.2.1 Flow Control

The flow control mechanism has to ensure two things: firstly that the multiplexor routeing, as a whole, can operate within a fixed amount of memory, i.e. a finite number of buffers (*deadlock free*); and secondly that all messages will be eventually delivered, regardless of the other traffic in the multiplexor (*livelock free*).

The algorithm adopted for the implementation of the deadlock- and livelock-free routeing is based on that of Toueg and Ullman [49], using a Forward State (F.S.) controller. A network is characterized by the longest route, k, that a packet

may travel, in terms of hops, from its source to its destination. A node, v, in the network has b buffers, giving a general description of a controller for the network as FS(b, k). The state of node v is described by j, where

$$\mathbf{j} \quad = \quad < j_0, j_1, \ldots, j_k >, \tag{5}$$

and where j_i is the number of messages in v that are i steps from their destination.

It has been shown [49] that, provided $b > k$, a FS(b, k) controller is deadlock free if it bases its acceptance of a message j hops from its destination on the set of inequalities:

$$\begin{aligned} \{\,(\mathbf{j}, j) \;:\; & \forall i \in \{0 \cdots j\}, \\ & i < b - \sum_{r=i}^{k} j_r \quad \wedge & (6) \\ & 0 \leq j \leq k \quad \wedge & (7) \\ & 0 \leq \sum_{r=0}^{k} j_r \leq b - 1\,\}. & (8) \end{aligned}$$

Condition 7 states that only messages of a state for which the controller has been dimensioned may be accepted. Condition 8 is a simple statement of the conservation of the buffer resource: a packet may only be accepted if there is a buffer available. It is straight forward to show that the upper bound of condition 8 is simply a special case of condition 6 with $i = 0$. Condition 6 expands to $j + 1$ inequalities, all of which must be satisfied if the packet is to be accepted by the node.

Since message passing is effectively a background task for the transputer (the main task is simulation), an important feature of any implementation of the flow control mechanism is that the processor should not "busy wait" when a message fails to meet the acceptance criteria; instead, it should block by descheduling the thread until it is permissible to proceed.

By labelling the inequalities generated from condition 6 for a state j message as I_0 through I_j, the acceptance function for a state j message, Acc(j) is

$$\mathrm{Acc}(j) \;=\; I_j \wedge I_{j-1} \wedge \cdots \wedge I_0$$

and

$$\mathrm{Acc}(j-1) \;=\; I_{j-1} \wedge \cdots \wedge I_0.$$

Hence, expressing recursively

$$\mathrm{Acc}(j) = \begin{cases} I_j \wedge \mathrm{Acc}(j-1), & \text{if } j > 0 \\ I_0, & \text{if } j = 0. \end{cases} \tag{9}$$

It is clear, therefore, that if, for some r, I_r would not hold true if the message were accepted, then all messages of state $j \geq r$ would fail on the same inequality. Further, if the subset of inequalities

$$\{\, I_x : x \in \{r+1 \cdots j\} \,\} \tag{10}$$

have already been satisfied, they need not be re-tested provided that no further messages of state $\geq r$ are accepted before the current one. (The value of the summation may only decrease due to transmission of messages that already have a buffer.)

It is important to note that a potential race condition exists: as expressed in condition 6, all the inequalities need to be tested in an unbreakable sequence; but a single semaphore cannot be used to guarantee this atomicity since this would lead to 'busy waiting'; to permit other messages to have fair access, each time a test fails the semaphore would have to be released, and then reclaimed, before testing again. Not using a semaphore, however, would lead to the race condition: two messages contending for access may test an inequality and find it valid when only one of them should have done so (which one is not important until livelock avoidance is being considered); one of the messages would then be granted a buffer under incorrect conditions, leading to the possibility of deadlock later on. Consider, as an example, a state j message, m_j, and a state l message, m_l, where $j, l \geq s$ and both are trying to gain acceptance to the node at the same time.

Assume, for inequality I_s that

$$b - \sum_{r=s}^{k} j_r - s = 1,$$

then either m_j or m_l might be accepted, but not both. If m_j were initially selected and passed all the tests, but before the state table, j, could be updated, m_l were also to test the same inequality, then it too could (incorrectly) pass all the tests when it should have failed at least one: a message would then be allocated a buffer to which it was not entitled and deadlock might follow.

To prevent the race condition, each of the inequalities is modified so that it now sums, not the number of messages accepted by the node, but the number accepted by itself, z_s. If the order of testing is arranged to start with the inequality representing the highest state and to progress towards the lowest then the two summations are related by

$$z_s = \sum_{r=s}^{k} j_r + \sum_{r=0}^{s-1} \text{count}(\{\text{ Blocked messages at } I_r : \text{state(msg)} \geq s \}). \qquad (11)$$

It can be seen that if $z_s > \sum_s^k j_r$ then the message acceptance would have subsequently failed when testing some other inequality; otherwise, z_s is degenerate to the original form.

Maintaining z_s is simply a matter of incrementing the summation after the test has succeeded, and decrementing it when the message is subsequently transmitted. To avoid race conditions it is now only necessary to ensure that the test-and-increment of each individual inequality is atomic. Condition 6 can now be rewritten as

$$\begin{aligned} i &< b - z_i \\ \Rightarrow 0 &< b - i - z_i. \end{aligned} \qquad (12)$$

Since each inequality should cause the message to block when the test fails, condition 12 can be implemented by a semaphore[2] with the initial value $b - i$.

[2]The n-value semaphore is a locking mechanism that permits at most n program threads to have access to a *critical section*, whilst any additional threads are forced to wait until another thread releases its lock. Semaphores are a generalization of token schemes where access is only granted to the holder of the token. In circumstances where the order of granting access to those threads forced to wait is important, the term *fair semaphore* will be used to indicate that access is granted using a first-come first-served scheme.

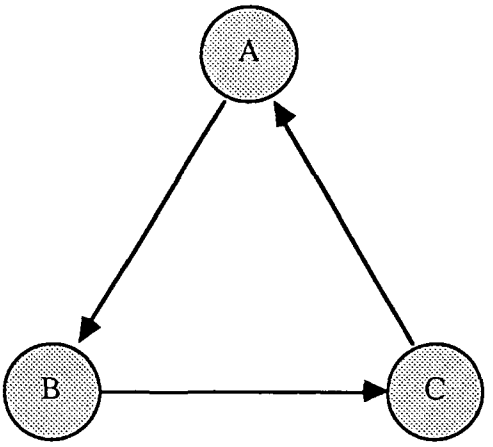

Figure 6: A simple network that can lead to deadlock with message passing. More buffers must be made available to messages nearer to their destination than to messages that still have further to travel.

3.2.2 Implementation of Flow Control

The entire state of the controller can be represented by an array of $k+1$ semaphores with initial values of b, $b-1, \ldots, b-k$; the acceptance algorithm consists of starting with the $j+1^{\text{th}}$ semaphore and working downwards, obtaining a lock on each one. When a buffer is freed, after the message has been transmitted, each of the semaphores waited upon is signalled.

To allow the flow-control mechanism to have complete control over the order in which messages are accepted, a node currently holding the message cannot forward it until the receiving node can guarantee its acceptance; it does, however, forward information stating that a message of a particular state is ready for transmission. To understand fully why this is the case, consider the rather contrived network of figure 6 in which each node passes all its messages in a clockwise direction, regardless of the destination to which it is intended. Each of the nodes generates messages to its anti-clockwise neighbour, necessitating that each message pass through all of the processors *en route* to its destination. Assuming that there are three buffers on each processor (the minimum, since the maximum path length is two), then all the buffers may be filled with state 0 messages, but at most two of them may contain state 1 messages and if one of them contains a state 2

message then not more than one of the remainder may contain a state 1 message. Given time, state 0 messages will clearly be absorbed (since these messages have reached the destination node), guaranteeing that, eventually, there will always be at least one free buffer in each node. If the sending node is permitted to select which message is transmitted then a cyclic deadlock can occur very rapidly. For example, each node receives a state 2 message from a source, leaving each node with two free buffers that may be used to hold two state 0 messages or a state 0 and a state 1 message; however, no more state 2 messages may be accepted at this time. Each node is able to forward the message to the next node in the chain, whereupon the messages are promoted to state 1, enabling another state 2 message to be accepted by each node. In this situation the remaining buffer in each node may only contain a state 0 message; this means that the state 1 message in the preceding node must be forwarded *before* the state 2 message. To prevent livelock we must guarantee to accept the state 2 message sometime; so it is not sufficient always to select the message at the highest priority.

For the receiving node to be able to determine when a message of a particular class may be accepted it needs to be told by the sending node that there is a message of this class available for transmission; when a buffer of the appropriate state becomes available a reply is sent and the message is transmitted: care needs to be taken when there are multiple links that races cannot occur between two or more links over the allocation of a single buffer; for this reason semaphores are used to provide an atomic 'test-and-allocate' routine. Figures 7 and 8 show details of the program threads running. When a node receives a buffer of a particular class (greater then 0) then the output link is selected and a request is generated to the next node in the chain; upon receipt of this request the receiving program thread initiates a further program thread to perform the acceptance control described above (this enables the receiving thread to return immediately, and without blocking, to its primary task of receiving messages across the link). The acceptance thread performs its work by waiting on the appropriate set of semaphores; when a 'lock' has been obtained upon each one it generates a 'reverse

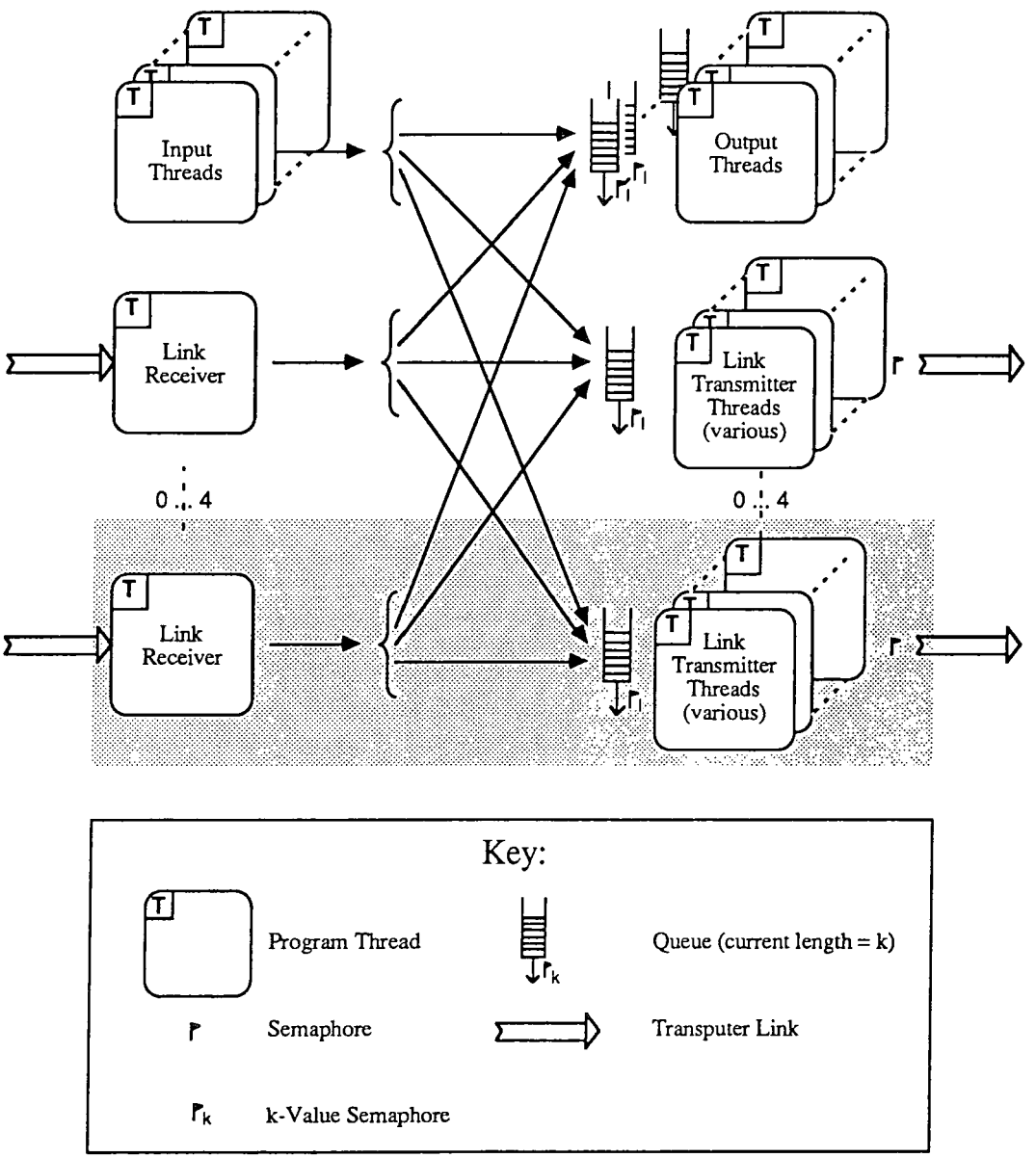

Figure 7: Threads running in a multiplexor process. The input and output threads receive messages from tasks on the local transputer; the link threads communicate with other multiplexors via the transputer's links. The grey-shaded area represents the threads associated with a single link and is shown in more detail in figure 8.

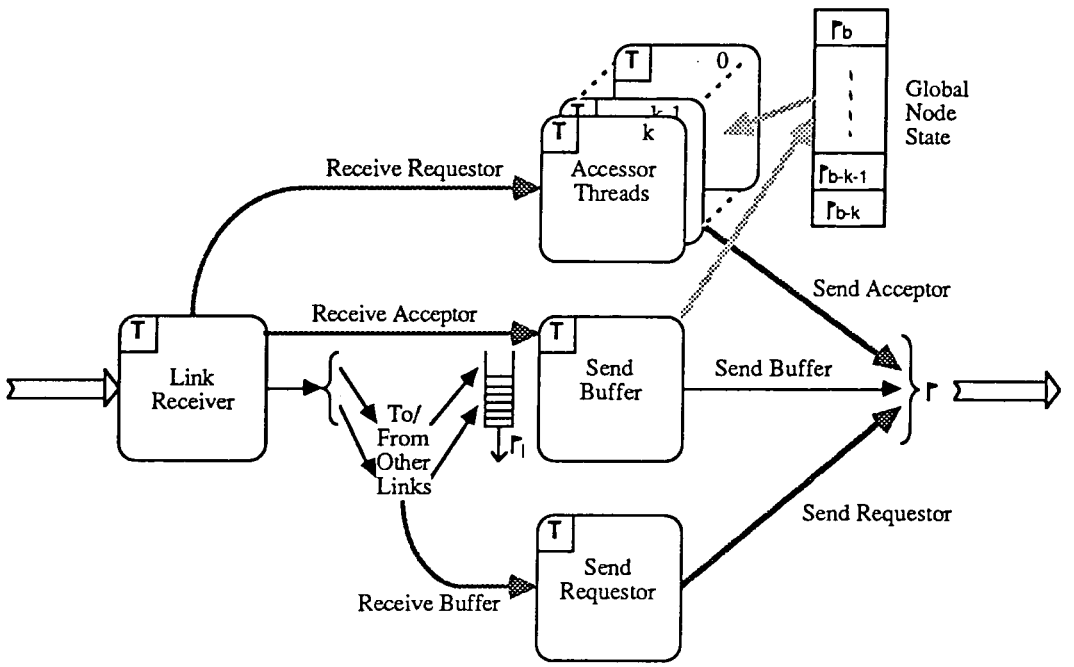

Figure 8: Detail showing the threads associated with each input-output link pair in the multiplexor.

message' across the link, back to the sending node, indicating that a message of the appropriate class may be transmitted. Again the 'receiving program thread' initiates a further thread to perform the transmission so that it is impossible for the two 'receiving threads' to be trying to send messages to each other simultaneously. Once the message has been transmitted, both the buffer and the semaphore locks that were claimed before allocating it are released.

3.2.3 Livelock Avoidance

Livelock avoidance is achieved naturally by the algorithm, provided the semaphores used are all fair (i.e. they grant access on a strictly first-come first-served basis).

Theory The implementation of the FS(b, k) controller is livelock free, provided that the semaphores used are fair.

Proof Assume that all state i messages are livelock free. Hence, a state i message waiting for access to a node must be able to claim each of the semaphores representing the inequalities derived from $z_i, z_{i-1}, \ldots, z_0$ (equation 11 and condition 12) in a finite period of time. A state $i+1$ message needs in addition, and first, to satisfy the inequality derived from z_{i+1}. If this first semaphore blocks acceptance of the message then there exists at least one message of state $i+1$ that has already achieved a lock on that semaphore (because the initial value of the semaphore is set to one greater than the inequality derived from z_{i+2}): such messages are then treated in an identical manner to state i messages and must consequently obtain locks on all the remaining semaphores; having done this they seek acceptance as state i messages at a remote node, and state i messages are livelock free. Therefore, a state $i+1$ message that has claimed a lock on the first semaphore is livelock free; since the semaphore is fair any state $i+1$ messages that are waiting for a lock on the first semaphore must be granted access in a finite time. Hence, state $i+1$ messages are livelock free. State 0 messages are clearly livelock free since they are at their destination and must be consumed in a finite time; so, by induction, all messages are livelock free. □

3.3 The Packetizer

Messages between the simulation tasks commonly consist of several small pieces of information: for example, a cell has associated with it not only the time of transmission and the data and header fields but also the time of creation, the size of the data field in use (for efficiency) and an optional series of trace-information packets that can be used when debugging the simulator. If each item were to be transmitted individually across the processor network then the efficiency of the multiplexor would be extremely poor; each packet in the multiplexor would contain perhaps as little as four bytes of information and an overhead of eight bytes (four bytes for each of the packet-header and the packet-size fields). To overcome this inefficiency, each simulation message (e.g. a cell) is concatenated

into a single packet (or a series of packets if this would exceed the maximum size of a single multiplexor packet) which is then transmitted to the receiving process.

In addition to the inefficiency associated with using the multiplexor with small units of data there would also be an overhead due to the establishment of the the occam channel for passing data between one task and the next. Each communication requires that both ends (the sender and the receiver) are ready to proceed before any data can be sent: if one end is not ready the other task blocks whilst waiting. Because of the way in which the transputer's process scheduler works this can mean a large number of process switches, each switch having a small overhead in terms of C.P.U. time; more importantly, each time a process is descheduled it is placed at the back of the relevant queue (either high or low priority) and has to wait its turn for further access to the C.P.U.. It is clearly more efficient if the number of times a channel communication has to be initiated is kept to a minimum; work by Gould *et al.* [50] has shown that the throughput of the channels increases dramatically as the size of the data block is increased.

3.4 The Synchronization Mechanism

As mentioned in chapter 2, the correct choice of synchronization algorithm is critical for optimal performance of the simulator: such a choice can only be made when some aspects of the system being modelled are known.

3.4.1 Characteristics of A.T.M. Networks

A more comprehensive review of Asynchronous Transfer Mode (A.T.M.) networks and protocols will be given in chapter 4, but a few aspects of the protocol that are relevant to the synchronization model are given here.

A.T.M. networks use a fixed-size data packet, known as a cell, which consists of 48 octets of data and 5 octets of header, giving a total cell size of 53 octets. They are typically transmitted, within the network, using multi-megabit-per-second media, such as fibre-optic links; such links will usually be running at data rates

in excess of 150 Mbit/s. It is possible to derive a simple formula that describes the number of cells that will be in transit across a link of a given length at any one time (the link can be considered as a delay line):

$$N = \frac{LSn}{lc}, \tag{13}$$

where L is the length of the link, S is its speed (adjusted to account for overheads such as framing), n is the refractive index of the transmission medium (typically, about 1.5 for a glass fibre), l is the cell size and c is the speed of light. Considering, for example, a 15km link running at 150 Mbit/s, then there may be up to twenty-six cells in transmission across the link at any time; longer, or faster, links would have correspondingly larger numbers of cells in transit. This 'pipe line' can be used to advantage as a method of look-ahead within the simulator.

3.4.2 Implementing the Available Look-ahead

In conventional packet-switched networks the delay between a bit in a packet being transmitted at the start of a link, to its arrival at the end of the link is insignificant when compared to the size of the packet and the speed of the link. For distributed simulators, therefore, it is of little importance to consider in detail whether the packet is transferred from the source to the destination at the departure time, or the arrival time. For A.T.M. networks this is no-longer the case.

If the cells are held in the source module until they are scheduled to arrive at the destination in the simulated system then valuable information about the future state of the destination's input queue is being withheld and the destination process is unable to make efficient decisions about what may be happening next. The converse situation is to transmit a cell immediately that it becomes ready and to queue it at the destination until its arrival time is reached: this gives more information about the processing order within the destination itself, but does not provide any further information about when the next cell will arrive on each link. Both these approaches use an assumption, which appears regularly in

implementations of the Chandy-Misra model, that the departure and arrival times of messages is the same in simulated time; for example, De Vries states [13] "The send time of a message is also the receive time of the message at the next process." For an A.T.M. network it is possible to relax this assumption to read 'the receive time of a message at the next process is the send time of the message *plus a constant delay.*' The constant delay may be non-zero provided that the behaviour of any items being represented by the message is deterministic (independent of all external influences) during the period of the delay.

The net result of the change to this assumption is that whilst a message may depart from the source process at a certain point in simulation time, it is not needed at the destination process until a known later point in time. The message is dispatched from the source process at the time that the cell is transmitted; as soon as it arrives at the destination the destination process knows the entire history of the link up to the scheduled arrival time of the cell (since the link is deterministic and cells cannot change order whilst in transit across it). Effectively, the destination can see a small amount of future behaviour for the link: this can be exploited for two ends; the avoidance of deadlock with fewer NULL messages and the improvement of concurrency between the processes.

3.4.3 Local Simulation Time

Given that messages can have different time stamps at their points of departure and arrival, it is now necessary to show that the simulation time of each local process can fluctuate, relative to its neighbours, within the bounds given by the time difference of the message's departure and arrival times. Further, it can be shown that the constant may be local to an individual connexion and need not apply to the network as a whole.

Two processes, P and Q, which are at simulation times T_P and T_Q respectively, are synchronized if their simulation times satisfy the inequalities

$$T_Q - T_P < t_{PQ} \tag{14}$$

$$T_P - T_Q < t_{QP}, \tag{15}$$

where t_{PQ} and t_{QP} are the constant delays associated with messages passing from P to Q and from Q to P respectively. Process Q is restrained using (14) and P using (15). That this is the case can be seen by considering a message leaving P for Q at time T_P: the message arrives at Q at $T_P + t_{PQ}$ which lies outside the bounds of (14) and hence cannot affect the current state of Q. It is trivial to show that the reverse also applies. If an inequality would be violated by further simulation of a particular process then that process must block.

Unfortunately, it is not possible for Q to know the exact value of T_P, or for P to know T_Q, since this would require the times to be global values. Instead, a lower bound on the time at a remote process can be maintained (for example Q would maintain the value $T_{P,Q}$ for its estimate of T_P) that is sufficient to ensure that the appropriate inequality can never be violated. $T_{P,Q}$ is simply the time at which the last message from P was transmitted to Q; it is maintained by both P and Q; P is able to calculate when $T_P - T_{P,Q}$ would otherwise exceed t_{PQ} and cause a NULL message to be sent to Q. Similarly, Q must block if $T_Q = T_{P,Q} + t_{PQ}$ until a message is received. This corresponds to the situation where the link becomes entirely quiet along its whole length: it can be seen that the number of NULL messages on a link cannot exceed one message for every N cell transmission intervals and that as the cell arrival rate increases the number of NULL messages drops rapidly; if the cell arrival rate were Poissonian then the number of NULL messages would drop as a negative-exponential function of the load.[3]

One final constraint is required to ensure that deadlock is not possible under any circumstance: the transmission of NULL messages must take priority over blocking, given that the two events are scheduled for the same time. While this condition is unlikely to occur whilst the simulator is in the main phase of the

[3]The cell arrival rate could never be a perfect Poissonian function since this would imply events being able to occur at infinitesimally close, but non-overlapping, times: since the cells are of finite size a better function for describing traffic on the link would be the output of a M/D/1 queueing process; such a system would require slightly fewer NULL messages since the queue acts as a smoothing function.

simulation, it occurs quite often during the initial transition phase when a link has not yet had a cell transmitted and when $t_{PQ} = t_{QP}$.

3.4.4 Managing Multiple Links

In extending the above discussion to a multi-process network with arbitrary interconnexions in a sparse (i.e. not fully connected) topology, two main areas need to be considered: firstly, that the synchronization remains valid between two unconnected processes which communicate indirectly using a third, and possibly subsequent, process; and secondly that a network of processes also synchronizes correctly and can never deadlock. Given these two proofs it is possible to build any arbitrary topology of processes that are synchronized; a fully-connected network is simply a special case.

Consider a three process network, PQR, where Q is connected to both P and R, but P and R can only send messages to each other using Q as an intermediary. Each process maintains its own local time, namely T_P, T_Q and T_R.

Theorem If P is synchronized with Q and Q is synchronized with R then P is implicitly synchronized with R. i.e. no event occurring at P can cause a causality error at R, and *vice versa*.

Proof P can only communicate with R by sending messages to Q: since P and Q are synchronized then there can be no causality error when the message is processed at Q; after this the message is processed in the same way as messages from Q to R. Q is synchronized with R so any messages from Q to R cannot cause causality errors at R. Hence messages from P to R cannot cause causality errors. The reverse is trivial to show; hence, P and R are implicitly synchronized. Mathematically:

$$\begin{aligned} T_Q - T_P &< t_{PQ} \\ T_R - T_Q &< t_{QR}, \end{aligned}$$

hence, summing the two inequalities

$$T_R - T_P < t_{PQ} + t_{QR}.$$

Any message from P to R will take $t_{PQ} + \delta Q + t_{QR}$ time to get from P to R, where δQ (≥ 0) is the time taken to process that message at Q: since this is at least as great as the time difference between them there can be no causality errors. □

Theorem If P is synchronized with an arbitrary, synchronized, sub-network, S, and S is synchronized with R then P is implicitly synchronized with R. i.e. no event occurring at P can cause a causality error at R, and *vice versa.*

Proof Since S is synchronized we may consider two of its access points, S_1 and S_2 (which may be discrete points in the sub-network, or the same node if S is a trivial sub-network): by definition S_1 and S_2 are synchronized, so messages arriving at S_1 can be forwarded to S_2 without generating a causality error; in doing so, they encounter a (possibly variable) delay of δS ($\delta S \geq t_S$, where t_S is the minimum propagation delay between S_1 and S_2). If P communicates with S using S_1, and R with S using S_2, then since P & S, and S & R, are synchronized,

$$\begin{aligned} T_{S_1} - T_P &< t_{PS} \\ T_R - T_{S_2} &< t_{SR} \\ T_{S_2} - T_{S_1} &< t_S, \end{aligned}$$

hence, summing gives

$$T_R - T_P < t_{PS} + t_S + t_{SR}.$$

A message from P to R will encounter a delay of $t_{PS} + \delta S + t_{SR}$ which is at least as great as the time difference between the two end points, ensuring that no causality error can occur. The reverse is trivial, since the network is symmetric, hence P and R can be shown, by induction, to be synchronized. □

Theorem In a network of synchronized processes at least one process is always able to proceed.

Proof The solution for one- and two-node sub-networks is degenerate to the initial situations covered earlier. For networks of three, or more, nodes closed loops can occur and it is necessary to show that these cannot deadlock. It has already been shown that a multi-node route has associated with it implicit synchronization

between the end processes. Consider the nodes A and C in the three node network shown in figure 9: there is explicit synchronization across the link AC and implicit synchronization using the route ABC.

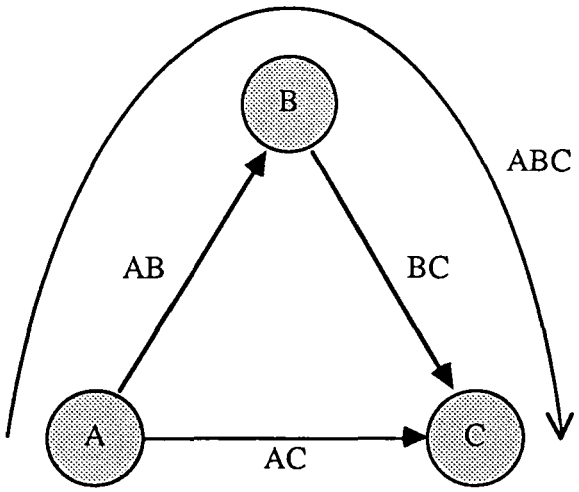

Figure 9: Simple network with explicit and implicit synchronization routes.

The proof proceeds by considering the dependencies between the blocked processes and showing, by *reductio ad absurdum*, that such a chain can never form a loop. This leaves at least one process that cannot be blocked by any other, and which must be able to proceed.

Assume that there is no node able to proceed. Consider a node, J, in the network that is blocked at time T_J: this node is unable to proceed since it is waiting for a message from a remote node, say K, which is at time T_K. T_K cannot be greater than T_J since otherwise J would not be blocked; further, if $T_K = T_J$ then some message must be generated from K to J, since the two processes are synchronized, and on receipt J will be able to continue (because for J to be blocked at time T_J, the last message from K to J must have been at time $T_J - t_{KJ}$): therefore, $T_K < T_J$. Similarly, due to the time constraints, K must be blocked on a node that has not yet been considered. This is repeated until a node is found where all its neighbours are at times greater than itself (in the limit this is the only node not yet considered): this node cannot be blocked by any other process and, hence, must be able to proceed. □

3.5 The Event Manager

The event manager is the heart of each simulation process: it maintains a list of pending events along with the simulation time when the event should be scheduled for execution. Rather than give the event manager detailed knowledge of the type of each event, all events are given a common form: a C function that takes a time value (the execution time) and an arbitrary pointer to the function's data space; the function returns the time at which it should be rescheduled, or the negative number `NOEVENT` if no rescheduling should take place. The functions are managed by the event scheduler as pointers to the function that should be executed. In addition, each scheduled event is given a priority (default 0) which can be used when two events are scheduled at the same time, i.e. highest priority first; events at the same priority will be processed in arbitrary order if their times are identical.

Routines are available to schedule new events at either the default, or some arbitrary priority; once the priority is set it persists across rescheduling. A routine is also available to remove events that have previously been scheduled.

3.5.1 Event List Management

The event list is normally maintained using the twin list method [44], but it is possible to convert the procedures to be functionally the same as a single list manager by setting the initial length of the first list to infinity. It was found that for the T4 series of transputers (which do not support floating-point arithmetic in hardware), using the twin list method approximately halved the amount of time spent maintaining the event list, but for the T800 transputer (which does support floating-point arithmetic) the change was negligible; indeed, for some configurations, the twin list procedure was slower by about 0.5%.

3.5.2 Synchronizing the Event Managers

In the Chandy-Misra simulation model there is not normally an event processor in the classical sense. Instead, events are replaced exclusively by messages and the

order of processing is determined by selecting the message with the oldest time-stamp: there must be a message available from each incoming link in order to be able to do this; the absence of a message causes the node to block. In the A.T.M. Network Simulator an event manager is used; consequently, in addition to adding dependence on the link mechanisms to the code of the event manager, monitoring for messages would be inefficient. To overcome this, the synchronization routines are implemented as normal events that run in the same manner as all other events in the simulator: two events are required for each link to a remote process; these are a NULL-message generator and a process blocker.

The NULL-message generator runs on the output of a link: it compares the current simulation time with the time when a message was last sent to the remote process; if this is less than a propagation delay it simply reschedules itself to a time one propagation delay later than the time at which the last message was sent; otherwise, it must be exactly one propagation delay since a message was last sent, so a NULL message is generated to the remote process and the generator reschedules itself one propagation delay later. The process blocker compares the simulation time against the time when a message was last received across a link from the remote process; if this is less than a propagation delay then it simply reschedules itself for one propagation delay after the time the last message was received; otherwise it blocks the current process until a message is received and then reschedules itself accordingly. The process blocker appears to the rest of the simulation as a routine that takes just sufficiently long to execute that the process remains in synchronization with its neighbours; however, while blocking, it consumes no processing time.

3.6 Configuring the Simulator

For any simulation tool to be useful it must be capable of being run with a series of different configurations, the extent of which has to be borne in mind when the simulator is designed. For a truly flexible system it is not normally sufficient for

these to be parameters that are 'hard coded' into the simulator itself; instead, they should be made available from a separate file (or by interactive prompting) at the time the simulator is invoked. In the ultimate case, not only parameters such as load and various delays should be configurable, but also the entire topology of the network itself: this can require substantial effort being expended on making the simulator easier to use, but, consequently, significantly more powerful.

3.6.1 The User Interface

'Hard coding' simulation variables into the simulator has already been dismissed as highly undesirable: not only does the simulator become hard to use, requiring recompilation between runs, but maintaining the simulator code becomes extremely difficult since modules are being constantly updated and modified. Maintaining a single module that contains all the variable parameters is almost as bad since most of the drawbacks already mentioned still apply and, given a small amount of extra work, it is normally possible to improve the situation significantly.

The next easiest method of obtaining the simulation parameters is to read a separate file when the simulator starts to execute: the file contains information, in a predefined format, which the simulator can use to configure various variables and data structures. Maintaining the code is significantly simplified and performing multiple simulation runs now simply requires that the simulator is re-run with a different configuration file each time. The main problem associated with such an approach is that the configuration file often appears, to anyone who is not well versed with that particular simulator, as a highly terse table of apparently meaningless numbers; mistakes in generating the file are easy to make, but are hard to detect, and they may often lead to bad results that have to be discarded after several hours of processing (if the user is lucky then the simulator may spot a bad configuration file and reject it).

An alternative to reading the configuration direct from a file is to prompt the user for the information as the simulator starts up: this can lead to requests

such as 'Enter the propagation delay for the link between nodes 5 and 6:' to which the user can insert a value. The advantages of such an approach are its flexibility over the 'hard coded' variant and the fact that the prompts describe what the value entered refers to: the disadvantages lie in the tedium of entering, by hand, large numbers of variables, many of which may be the same for similar elements and for subsequent runs, and in the difficulty of making corrections when mistakes are made. Interfaces of this type often require that the run is aborted in such a case, making it necessary for all the values to be re-entered next time the simulator is run: the probability of entering all the values correctly clearly falls as the simulator becomes more complex.

A significantly better method is to use a parsable grammar that describes the simulation parameters (and some of their dependencies) in a human comprehensible format: in such environments it is rarely necessary for the information to be in a totally fixed order since each parameter will have a tag associated with it that describes it uniquely. Comments are normally easily supported. An example entry might contain:

```
link 5:                          % Link between nodes 1 and 6
     prop_delay = 100u S
     speed = 100M bitps
     ;
```

Parsers for grammars of this type are easily produced using tools such as `yacc` and `lex` and would, probably, be implemented using a pre-processor for the simulator that produces the configuration tables that the simulator itself reads. Another advantage of this approach is that default values can now be used: a special entry (for example '`link default:`') might contain a series of fields that should be used when a real definition omits a parameter.

Perhaps the ultimate solution is to use C.A.D. techniques and a graphical interface to allow the user to 'create' the network that is to be simulated. The simulation consists of elements such as nodes, links and traffic generators, which

are represented by icons; these can be added to the screen using a pointing device, such as a mouse, until the required network is obtained. As each element is added the default values appropriate to it are inserted: a data window containing these values can be opened, enabling individual parameters to be fine-tuned as desired. Many topological checks can be performed as the network is created, for example a traffic generator must be connected to a node, and a point-to-point link must be connected to exactly two nodes at all times. Commonly used features, such as the icons, are available from a permanent menu on the screen; less frequently used options can be made available from a series of pull-down menus that can be arranged in a hierarchical manner to reflect their relationships. Mistakes are easy to rectify: items can be inserted or deleted, and fields can be re-edited if a wrong value is inserted. A checkpointing (undo) facility can also be used to allow certain errors to be immediately rectified, provided no further changes have been made. Once the configuration is complete the information is saved in a format that is easily parsable by either the configuration routine or the simulator. Whilst still being a separate program, the simulator can be invoked, if desired, by selecting a menu option from the configurer; this obviates any need to return to the operating system, but makes it possible to invoke the simulator direct, using a set of previously prepared files, if a series of runs need to be performed without operator intervention. All the features described in this section (with the exception of the undo option) have been implemented by the author in a simulator interface program designed to work with a traditional packet-switched network simulator running on a single processor: extension of the system to work with a multi-processor simulator would either require additional information from the user to map the various elements of the simulation onto the processors, or would require quite complex algorithms and heuristics to do this automatically; such algorithms would require information on load balancing and other parameters of the simulator, and fall outside the scope of this thesis.

The A.T.M. Network Simulator currently parses two files when it starts to run: the first describes the topology of the network being simulated and how

the individual processes should be mapped onto the processors of the transputer network; the second contains the various parameters required by each individual process. Both files are of the 'table of values format'. A parser is available for generating the first file that understands a superset of the *3L configurer* language [51]; the extensions are mainly aimed at supporting the reconfigurability of the transputer array used. The second file has to be generated by hand, but a built in pre-processor parses the special symbols '%date' and '%seed', replacing them with the current date and a unique random-number seed respectively. The seeds are generated using a different random number generator from the one used during simulation in order to avoid, as far as possible, correlations between the random number streams.

3.6.2 Booting the Simulator

The compiler package supplied by 3L Ltd [51] consists of three main components for use with multi-transputer networks: the compiler, which produces object modules from the source files; a linker, which links object modules and libraries to create tasks; and a configurer, which binds several tasks together to form an executable application. A task is a program in its own right: it is allocated a stack and an area of memory, and has its own global variables; it must always run on one processor, but can spawn *threads* which execute part of the code of the task in parallel and share the memory (they each, however, have their own stack); a task can only communicate with other tasks by using the occam channels implemented in the processor hardware: the collection of program threads in a task are referred to collectively as a process. The configurer is responsible for allocating tasks to processors, creating initial stacks and heap areas, and for mapping the connexions between tasks onto occam channels (both internal and external).

Unfortunately the configurer supplied with the compiler does not support the link-switch mechanism in the transputer network used and, therefore, cannot be used in the traditional sense to boot the entire network. Instead, there are two approaches available to overcome this limitation.

In the first approach two applications are configured: the first runs on the fixed topology part of the network (the transputers to the left of the link-switches in figure 4), this sets the link switches to the topology used by the second application and then loads it into the network; the second application is configured for a particular topology in the traditional sense. This approach is straight-forward to implement, but has two main drawbacks: firstly the second application has to be pre-configured before the simulator is run (this comes very close to 'hard coding,' which was rejected as being undesirable in the last section); and secondly two configuration files are required simply to boot the network, the first to set the switches and the second to configure the application.

The second approach, which was used in the simulator, is to use a small main application, that runs on the fixed topology part of the network, and a series of un-configured tasks. The main application does on-the-fly configuration of the remainder of the application using a single file that describes the simulation run. To do this it uses the low-level configurer execution primitives that are documented in the manual to load the tasks directly into each processor. The procedure is, relative to the alternative, more complex to implement, but has neither of the disadvantages. The main drawback is the possibility that such an approach might not be available in future releases of the compiler package.

3.6.3 Loading the Simulation Parameter File

Once each task has been loaded and has started to run, the simulation parameter files have to be loaded. Unlike traditional simulators this poses a large problem: part of the information contained in the parameter file is used by the multiplexors to control the switching of messages; until this is loaded they cannot operate properly. Similarly, none of the other tasks knows any information about where it lies in the overall topology, since to provide this information would require 'hard coding'. Indeed, the only information that each process has is its own array of channels for use in communicating, but even this has little meaning unless some conventions are used. Fortunately, 'false' channels can be created

during the configuration process and their values set to represent something other than a genuine channel.[4] With this information, known as a 'tag', each task in the simulator can be uniquely identified, enabling it to extract the relevant information from the parameters file.

At this stage a task still does not know on which input channel it will receive the configuration information; further, it does not know on which output channels, if any, it must forward the information so that it can reach its neighbours. To obtain this information a boot-tree is built which starts at the task connected to the fixed topology part of the network (there is exactly one such task) and extends outwards until all the tasks know their parent, and any children they might have. The protocol for doing this in the presence of loops is quite complex if the use of timeouts are to be avoided; the petri-net in figure 10 represents the code running on just one channel pair of one task (all of the channels in the simulator are paired, one input and one output, to the same remote task), the same code runs on each channel pair throughout the simulator.

The parameters file contains a few lines of global information, such as the title of the simulation run, the size of the network, and for how long the run must last, followed by a series of entries, one for each task in the simulator. To avoid the need for each task to have to be able to interpret information for other tasks (which may well be of a different class), each task scans the parameters file looking for a string of the form '`class xxx:`', where the `class` is the type of task ('SRCE' for a traffic generator, 'MUX' for a multiplexor, etc.) and `xxx` is the tag-value that was bound to the false link. On finding this string, the task then interprets the following parameters as its personal configuration file. Special routines are used to parse the file while ensuring that at the same time the entire file is passed on to its children in the boot-tree without modification or

[4] A channel is implemented as a memory location that contains a pointer to the block of memory that is to be moved; the D.M.A. controller in the transputer hardware performs a block move when both ends become valid. Special locations exist for the external links, but the principle of operation is the same. The array of channels given in the parameters passed to `main()` when the task starts are simply lists of addresses of the pointers (i.e. pointers to pointers): false channels are created by using illegal addresses; instead of being an address, the item in the list is given a numerical meaning which can be interpreted direct.

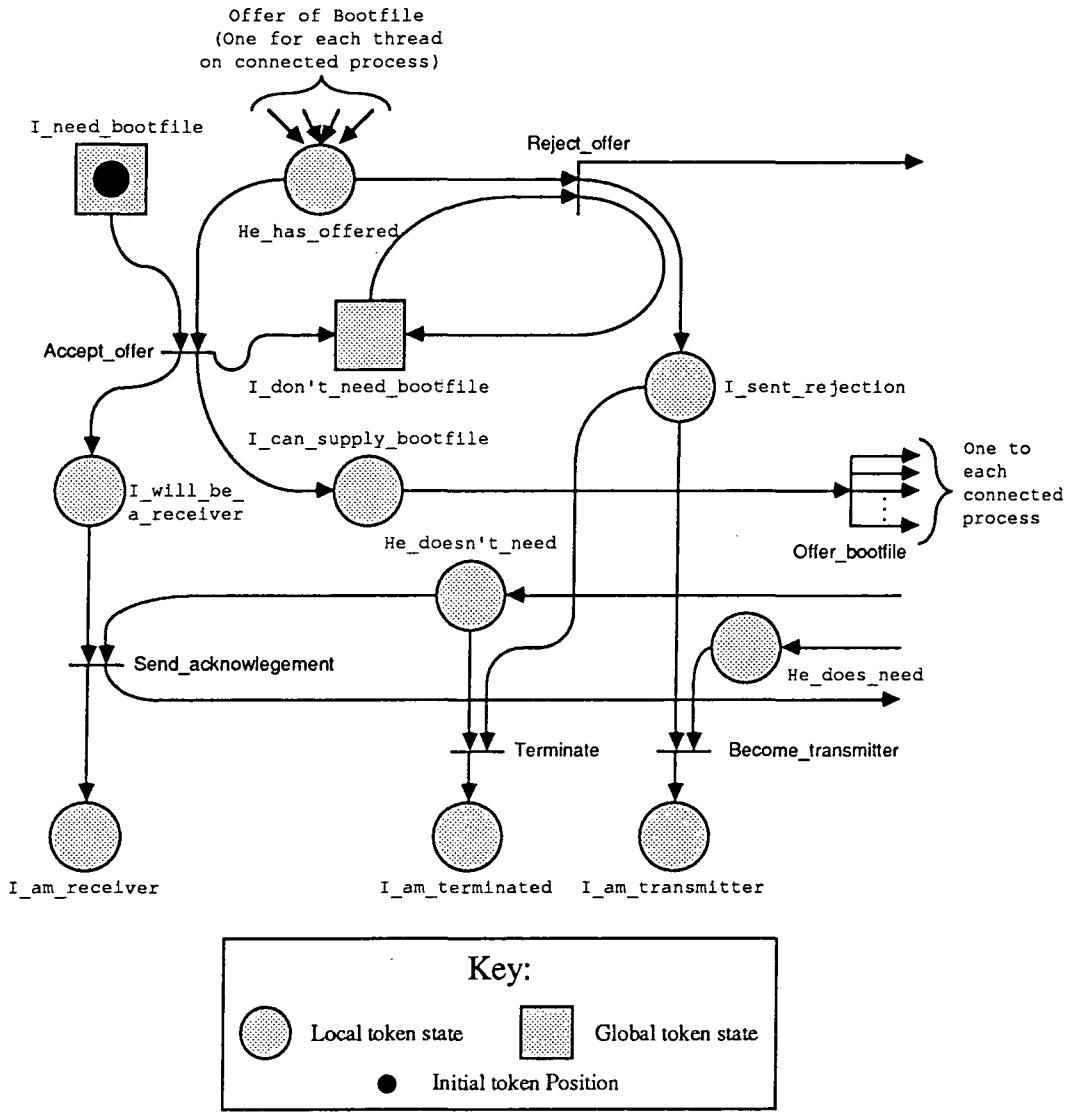

Figure 10: Petri-net showing the state transitions for a single channel while determining the download path for the simulator. The 'square' states are shared by all of the channels, which have been omitted for clarity, making it impossible for more than one channel to be activated as a receiver.

loss. Once the entire file has been read and interpreted, the configuration process is complete and the simulation can begin.

3.7 Performance Analysis of the Simulator

With distributed simulation, the ultimate goal is to obtain a simulator that runs as quickly as possible; if the speed of the distributed simulator is less than that of a conventional simulator then there is no reason for using it (and many good reasons for not doing so). However, it is normally impossible to compare directly distributed and conventional simulators since the two are written in an entirely different manner and the programmer rarely wants to write both. A good indication of the possible behaviour of the conventional simulator can sometimes be obtained, though, by running an optimized version of the distributed simulator on a single processor. The time taken for the single processor version to run can be compared with that for the multi-processor version and the *speed-up* of the simulator is then the ratio of the time for the multi-processor version to that for the single processor: normally this should lie in the range between one and n, when the multi-processor version is run on n processors; a speed-up of n is said to be *linear*.

The performance results given here are for the A.T.M. Network Simulator configured as shown in figure 11: the network consists of four A.T.M. exchanges in a fully connected trunk network and eight 'local' exchanges each of which is dual-parented onto two trunk exchanges; each local exchange has two traffic generators; the exchanges were all running the Orwell ring protocol (see chapter 5). Two sets of results were recorded using differing switch capacities and traffic mixes. In both cases the links were running at 150 Mbit/s and the propagation delay was set to 1×10^{-4} s (equivalent to about 20 km of glass fibre, or about 35 cells). The results for the very low traffic load were taken using 150 Mbit/s Orwell rings for the switches and with a mixture of voice and mobile traffic (see chapter 6); the results for the higher loads used purely voice traffic and a ring speed of

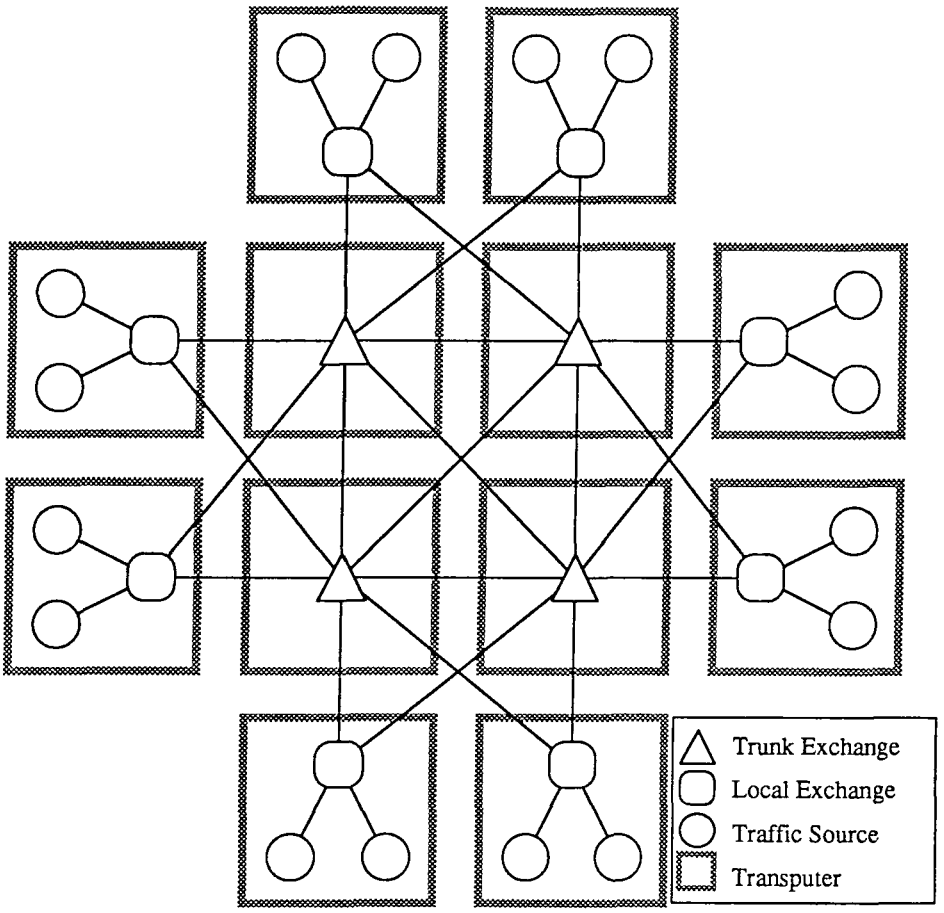

Figure 11: Network topology used for the simulator performance analysis runs. The basic processor assignments are also shown; a traffic source presents a very small load to a processor so it may be safely combined with a local exchange without unduly affecting the load balance.

600 Mbit/s. With the smaller capacity switches the maximum link loading was about 15%, but this was increased to about 50% for the high-capacity switches. Two single-processor simulations were run for each load: one with identical code to the multi-processor version; the other with the redundant multiplexors removed to speed message transfer. In the following graphs, when the load is shown it is expressed as the average percentage of the capacity of a link.

Figure 12 shows the time taken to simulate the two models on the full array of processors. The fact that the two curves do not pass through the origin has two causes: the NULL-message traffic for low loads and the overhead of simulating

the ring slot-rotation action for the Orwell protocol. That it is the second of these that represents the largest factor can be inferred from the fact that the NULL-message traffic generated for each of the two curves is identical for a given link loading; so for this to be the cause the two curves should cut the axis at the same point. Results similar to these, from work on a simulator for the Orwell protocol on a sequential machine, provided the motivation behind the work in chapter 5.

Figure 13 shows the speed-up of the simulator as a function of load; it shows that, even for a load of just 15% of maximum capacity, the speed-up is approaching the ideal value of 12 for the unoptimized version, and is starting to level out at just over 10 when compared with the optimized version. The difference between the two curves represents the proportion of the processing time that is taken up in switching the messages from one processor to another. The speed-up of the simulator relative to the unoptimized version can also be estimated from the CPU activity monitoring of each of the transputers in use: the results from doing this agree well with the upper curve shown. Figure 14 shows the speed-up for the 600 Mbit/s ring at the higher loads. In comparison with the unoptimized single-processor version the speed-up is greater than 10 for all loads simulated, and for link loads greater than 30% it is nearly 'ideal'.

It can be seen from figure 15 that the speed-up degrades gracefully with increasing NULL message ratio; but, fortunately, as can be seen from figure 16, the NULL message ratio remains very low for a large range of the load.

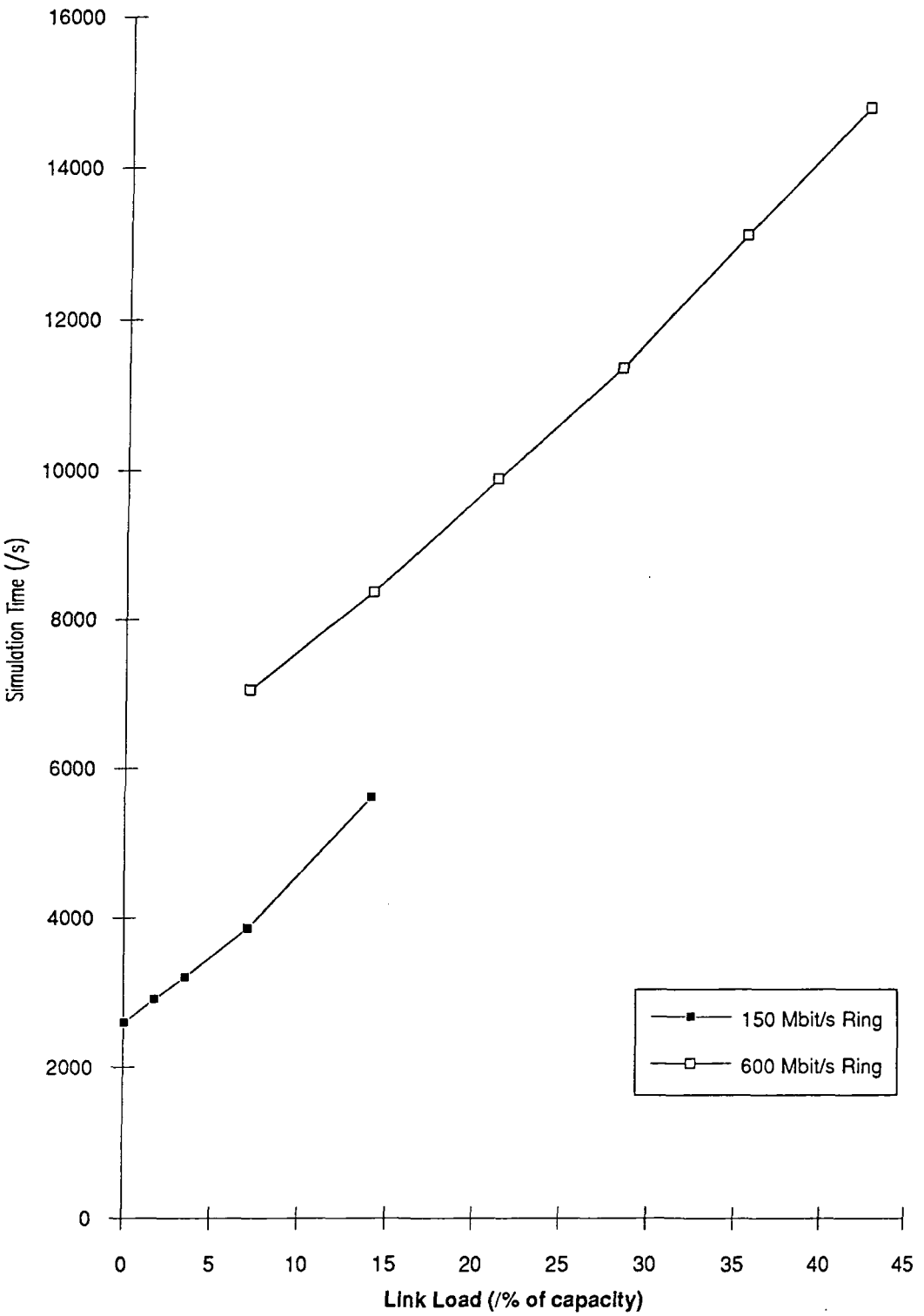

Figure 12: Simulation time for the twelve-node network on twelve transputers.

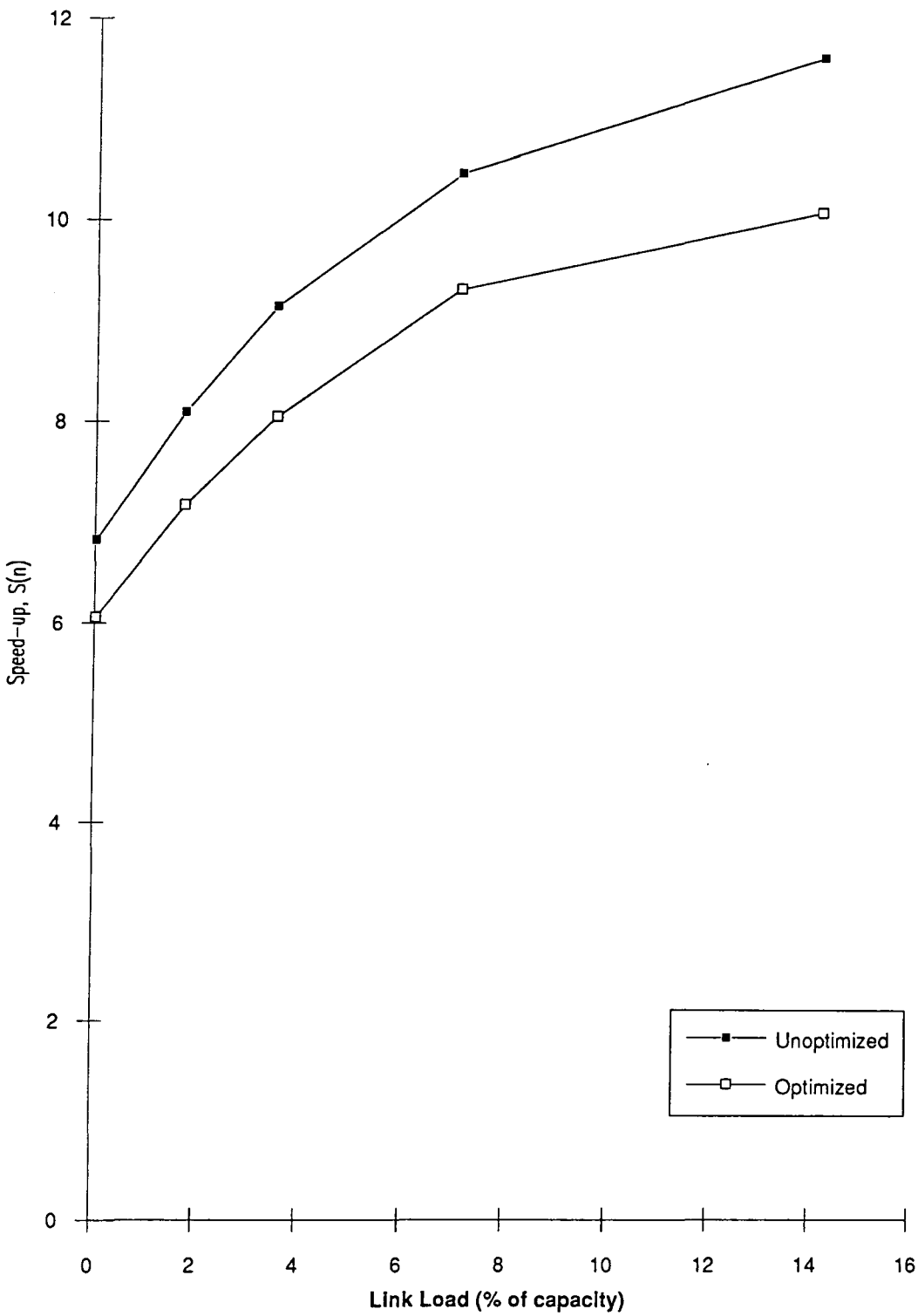

Figure 13: Speed-up for the 150 Mbit/s ring carrying mixed mobile and voice traffic. The Unoptimized curve is when the single processor runs identical code to the multi-processor version; for the Optimized curve the redundant multiplexors in the single processor version have been removed.

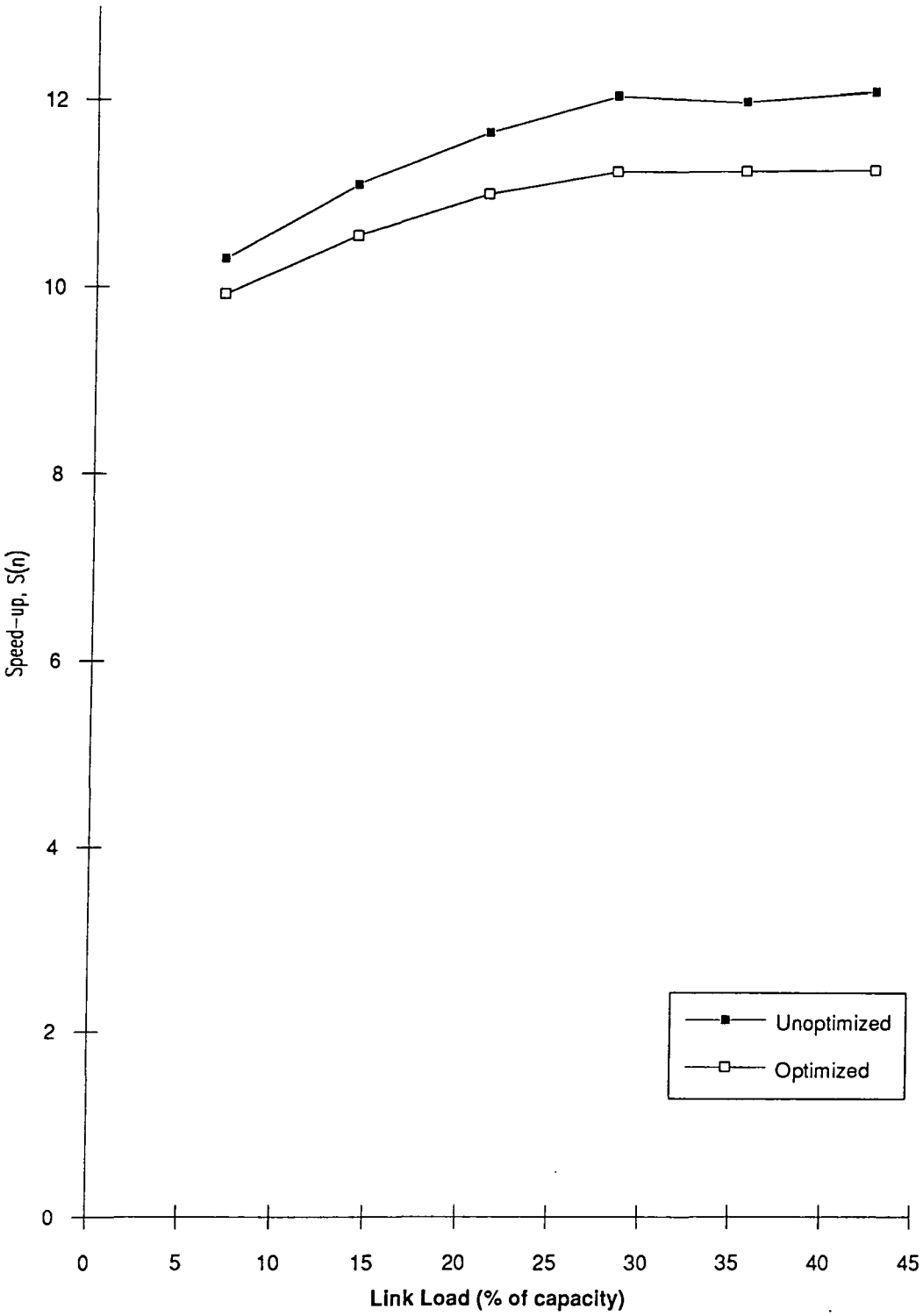

Figure 14: Speed-up for the 600 Mbit/s ring carrying voice traffic.

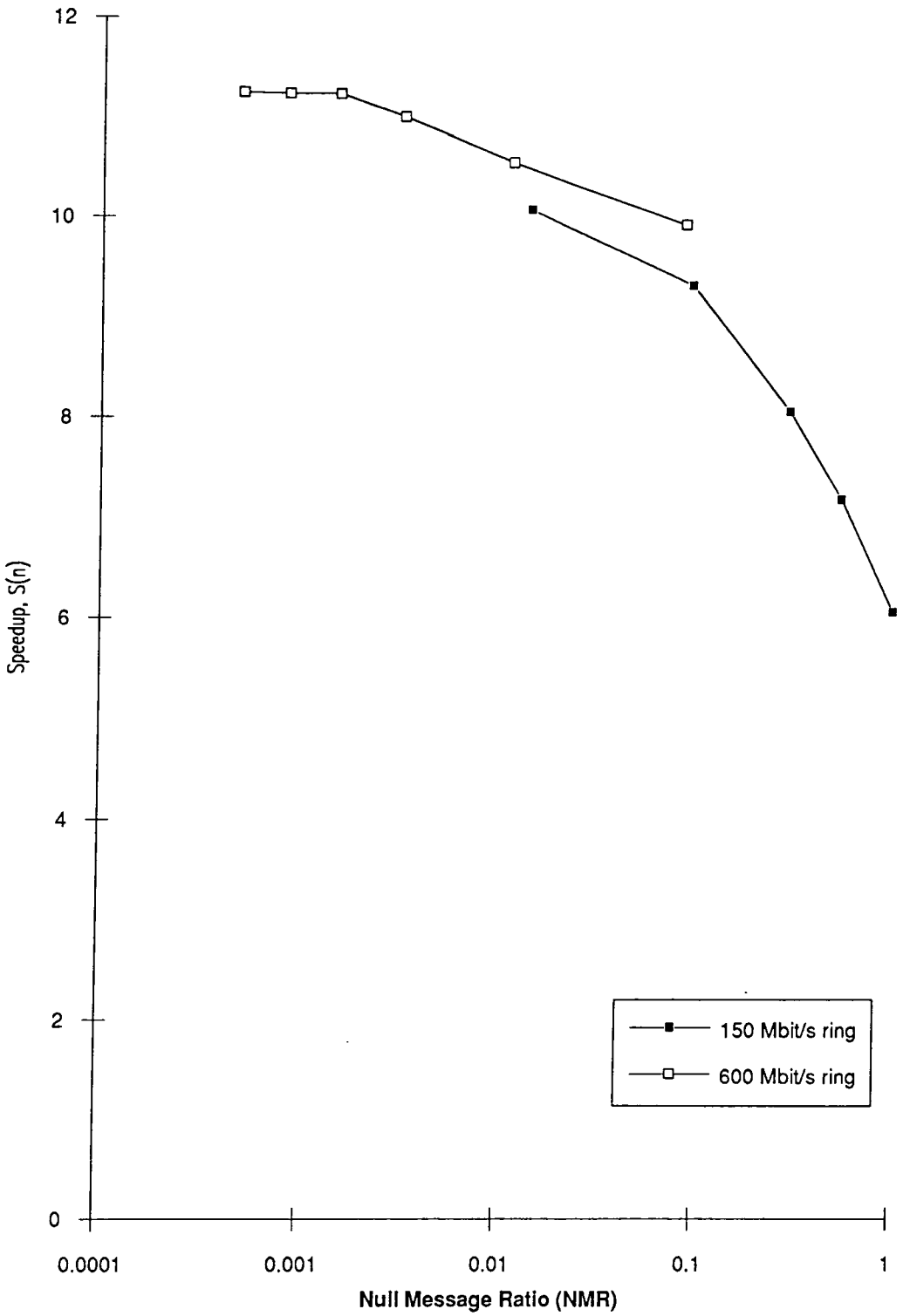

Figure 15: Speed-up as a function of NULL-message ratio. The difference between the two curves represents the extra parallelism that can be extracted from the higher speed rings.

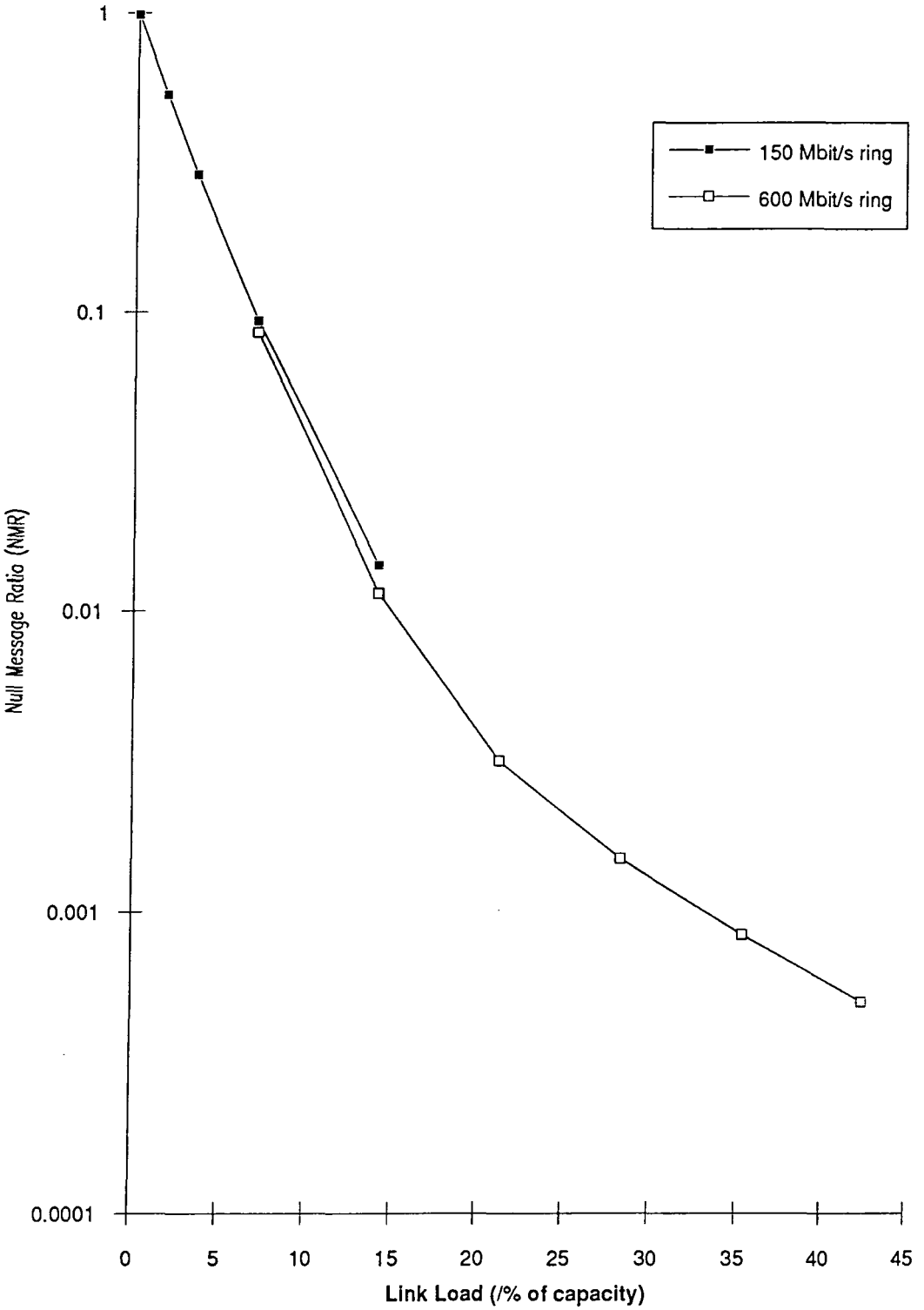

Figure 16: NULL-message ratio as a function of load. As might be expected, the ratio is independent of the ring speed.

Chapter 4

Asynchronous Transfer Mode Techniques

TELECOMMUNICATIONS STANDARDS have usually followed some time after the development of products for a particular need. This has led to subtle, or sometimes gross, incompatibilities between two functionally similar products, making interworking both inefficient and expensive. When telecommunication networks were largely considered as a local, or national, resource then the incompatibilities were not particularly significant; but, in many respects, the public network now needs more and more to be regarded as an international resource and such matters as interworking cannot be 'put on one side' until the local network is functioning. The cost of designing and installing the equipment is now so great that to justify replacing the current network with a new one requires that significant benefit be obtained by doing so and, further, that a similar need must not occur again for a long time (if ever).

The first telecommunication networks were entirely analogue and catered purely for voice communication; but the advent of digital electronics, particularly with development during the nineteen-fifties and -sixties of the transistor and the integrated circuit, made the eventual change to a digital communications network almost inevitable. By 1991, for example, the entire B.T. trunk network had been converted to digital switching and transmission, but substantial amounts of the local network, and in particular the final connexion to subscribers (the latter known as the copper pair), remain analogue; so far, it is primarily large users who have received direct digital connexion to the network. Digital networks, however,

can carry many more services than voice communications alone, the most obvious of which is computer data (for example, file transfer): such data networks, normally categorized as Local, Metropolitan and Wide Area Networks (LANs, MANs and WANs, respectively), usually have an inverse relationship between their capacity and coverage region.

Unfortunately, traditional digital voice networks and data networks use incompatible transmission techniques and have different performance criteria: voice telephony requires only moderate levels of data integrity, but has stringent constraints on the delay that may be incurred while passing through the network; for data communication, on the other hand, the constraints are the other way round. To resolve these differences voice communication uses synchronous techniques such as time division multiplexing (T.D.M.)[1] which has low latency at the exchanges since no queueing is involved: data communication is normally packetized; the traffic characteristics of the data being transmitted are hard to quantify in advance, so unlike voice, where a fixed amount of the communications resource can be allocated to each call, the data are split up into variable length packets and given access to the entire available bandwidth; transient overloads at the exchanges (switches) are handled by queueing.

It was clearly desirable for the services provided by the two systems to be merged into a single network using a single access point: the capital value of the local network is immense, and the cost of duplicating it, including the disruption this would cause, to provide data services is almost unimaginable. Fortunately, recent advances in technology have made it possible to use each copper pair in the access network (which was originally designed to support a single 3 kHz bandwidth channel) at a transmission rate of 144 kbit/s (sufficient for two 64 kbit/s voice channels plus some spare). In 1984 the C.C.I.T.T. published a series of recommendations (the I-series) that defined an *integrated services digital network*

[1]T.D.M. and its more analogue counterpart, frequency division multiplexing (F.D.M.), are classed as synchronous transfer mode (S.T.M.) techniques: the interpretation of the data received at the destination is based entirely on some pre-arranged time (frequency) allocation.

(I.S.D.N.) which were subsequently refined in 1988. The recommendations standardized a total of six different types of channel (labelled A to E and H), the most important of these being: the B channel, a 64 kbit/s channel for voice or data; the D channel, a 16 or 64 kbit/s channel for signalling or packetized data; and the H channel, which can run at 384, 1536 or 1920 kbit/s for higher-rate access. To standardize access further, two main channel combinations were defined: the *basic rate*, which contains $2B + 1D$ channels and is suitable for use with the standard copper pair; and a *primary rate*, which contains $30B + 1D$ channels in Europe and $23B + 1D$ channels in the U.S.A. and Japan.[2] The I.S.D.N. defines the B channel purely in terms of its size and position in the frame for transmission to the exchange (O.S.I. layer 1); no meaning is attached to any of the bits within the channel: this makes it suitable for use with any service requiring not more than 64 kbit/s bandwidth; services requiring more than this must use either multiple circuits or one of the H channels (the two are not interchangeable), but in either case the network does not 'see' any of this higher-layer protocol adaptation.

There are several problems, however, with the I.S.D.N. architecture that make it potentially unsuitable for many communication needs in the not too distant future. The internal workings of an I.S.D.N. do not naturally form a single network, once information reaches the exchange it is normally split into the appropriate data class and routed on the appropriate type of network (for example, voice traffic over a T.D.M. network and data over a packet-switched network); the B and H channels are highly restrictive in nature as they each have a fixed capacity that cannot be exceeded, whilst if a service requires less than the capacity of a channel the balance is wasted; in a similar manner, the suitability of the channels for packetized data is very poor, the D channel has very low capacity and the B and H channels do not support their stochastic traffic nature very efficiently. The addition of new services is also quite complex; this follows from the individual carrier-networks that tend to make up an I.S.D.N.; adding a new service may

[2]This is a classic example of the subtle interworking problems that exist when standards are defined to follow existing technology: the European configuration is compatible with the C.C.I.T.T. standard 2 Mbit/s connexions; the U.S. configuration with the frame structure of A.T.&T.'s T1 system, using 1.5 Mbit/s links.

well require a new type of network in addition to more software at each exchange. Finally, and perhaps most importantly, the I.S.D.N. does not define any access rate higher than 2 Mbit/s, this means that services such as high definition video cannot be carried. Most of these problems can be solved with the newly emerging standards for a *Broadband* I.S.D.N. (B.-I.S.D.N.); this defines both the interface and the network itself in a far more flexible manner which provides not only for much higher data transfer rates but also makes the installation of new services much simpler for network managers.

This chapter describes in detail the background and functionality of a B.-I.S.D.N., showing how both new and old, error-sensitive and delay-sensitive, services can co-exist on a single network. The prospects of implementing such a network are still a few years in the future, and it will not appear 'over night': in the interim it will be necessary for the new and old networks to interoperate. At present, the plain old telephony service (POTS) is still, by far, the largest revenue earner for the network operators and such a service must continue to be provided with an equivalent, or higher, grade of service (GOS, or quality of service, QOS) in any new system that is installed.

4.1 Background

When a broadband version of the I.S.D.N. was first proposed it was unclear what mechanism would be required to support the new services which it would provide. At that time most of the high-bandwidth services that would need to be carried by such a network (for example video based services) could only be encoded using constant bit-rate (C.B.R or F(ixed).B.R.) techniques, so it was suggested that the best approach should be to define yet more higher access rate channels, similar to the H channels: however, another of the main services that the network would have to support is high-speed connexion between LANs, which tends to have a bursty traffic characteristic. When it subsequently became possible to encode video signals using a variable bit-rate (V.B.R.) scheme it became

obvious that fixed channel allocations would be far too wasteful when segmenting the communications resource: further, for efficiency of channel allocation, it would probably be necessary to partition the S.T.M. frames in a fixed manner with each class of channel only being allowed to start on the appropriate boundary within the frame; this can lead to fragmentation, making it impossible to carry the higher-rate channels even when there is theoretically sufficient spare capacity.

It became clear that a new transfer mechanism would be required to make more efficient use of the capacity of the network. Two differing schemes were under investigation in the United States and Europe (with different applications in mind); it was proposed that the two should be analysed in more detail and a common solution adopted: the solution was initially referred to as the new transfer mode but was subsequently changed to asynchronous transfer mode (A.T.M.) because of the analogy with S.T.M.. The American proposal, fast packet switching (F.P.S.), was originally aimed at producing a high speed network suitable for connecting LANs together: the European proposal was more concerned with handling packetized voice samples in a delay-critical environment and is known as asynchronous time division (A.T.D.).

4.1.1 Fast Packet Switching

The C.C.I.T.T.'s X.25 and associated protocols were originally designed with relatively poor-quality copper-based transmission techniques in mind: the error rates are high, so the probability of a packet being transmitted from one end of the network to the other without some errors creeping in is not negligible; complex retransmission policies are employed across each link in the transmission path so that each packet is received correctly on one link before it is forwarded to the next; and windowing techniques are employed to enable sufficient throughput when the link delays are high. All these techniques require substantial amounts of software for verification and management of the protocol, and memory for buffering: as the reliability of the links increases to the extent that errors are

no-longer likely over the end-to-end path, the protocol becomes a hindrance to efficient use of the network. It has been shown that when the bit error ratio drops below 10^{-5} the use of end-to-end retransmission becomes no less efficient than link-by-link [52][3].

When fibre-optic links are used the bit error ratio can be reduced to levels significantly below 10^{-5}, so the proposal for fast packet switching involved the use of end-to-end retransmission. Moving error control and, also, flow control to the edges of the network significantly simplified the link protocols; the remaining functions were then easily implemented using hardware, resulting in extremely fast switches, and made it possible to carry delay-sensitive services such as voice. Since a primary service for the network was to be high-speed LAN interconnexion, variable sized packets were proposed.

4.1.2 Asynchronous Time Division

Whilst F.P.S. was being derived from traditional packet switching techniques, asynchronous time division was evolving from the synchronous time division (S.T.D.) techniques used for traditional circuit switching. By adding more intelligence to the switch and by adding short headers to each block of data, the need for time-position dependence was removed and the flexibility of the system enhanced. Research was performed to enable A.T.D. to be defined as a layer 1 protocol in the O.S.I. reference model; the functionality of the header was reduced to the absolute minimum required for switching the frame of data: fixed sized frames of between 8 and 32 octets were proposed, so the header needs to contain only connexion (routeing) and priority information. Because of its origins, A.T.D. was always intended primarily as a low-latency transport mechanism for delay-sensitive services such as voice and video, with the emphasis on video [54].

[3]This reference cites [53] for this result.

4.2 Asynchronous Transfer Mode

Rather than select one of the two alternatives, the C.C.I.T.T. elected to choose a compromise solution: some of the features of each of the proposals were chosen to form the A.T.M. implementation to be used for a Broadband I.S.D.N.. It was decided, for example, that fixed-size packets, known as cells (to avoid confusion with traditional packet switching, where the packets are of variable length), would be used, but their size would be longer than proposed in the A.T.D. definition; a small amount of extra functionality was added to the header over that proposed for A.T.D., but explicit priority information was omitted.

In O.S.I. parlance, A.T.M. provides layer 1 and some layer 2 functionality; the precise relationship between the A.T.M. reference model and the O.S.I. model is still to be fully defined (C.C.I.T.T. recommendation I.321) and, in any case, the mapping is fairly vague: many functions which have traditionally been part of the data-link layer are now omitted entirely or moved to functions in the transport layer. In traditional packet-switched networks the protocol stack at intermediate nodes in a packet's route covered layers one, two and three; for an A.T.M. network, however, the A.T.M. layer is the highest protocol layer within intermediate nodes. Figure 17 shows the layers within the A.T.M. protocol stack. The Physical Layer has two sub-layers: the physical media sub-layer, which is the lower layer and is responsible for bit timing and transmission; and the upper layer is the transmission convergence sub-layer, which handles aspects such as cell delineation and header error control. The A.T.M. layer has further sub-layers of its own, which include: Generic Flow Control, which only operates at the edges of the network and is used to throttle back sources when the network is in danger of overload; cell header generation and extraction; virtual-circuit and virtual-path number translation when switching the cells; and cell multiplexing or demultiplexing between supported services.

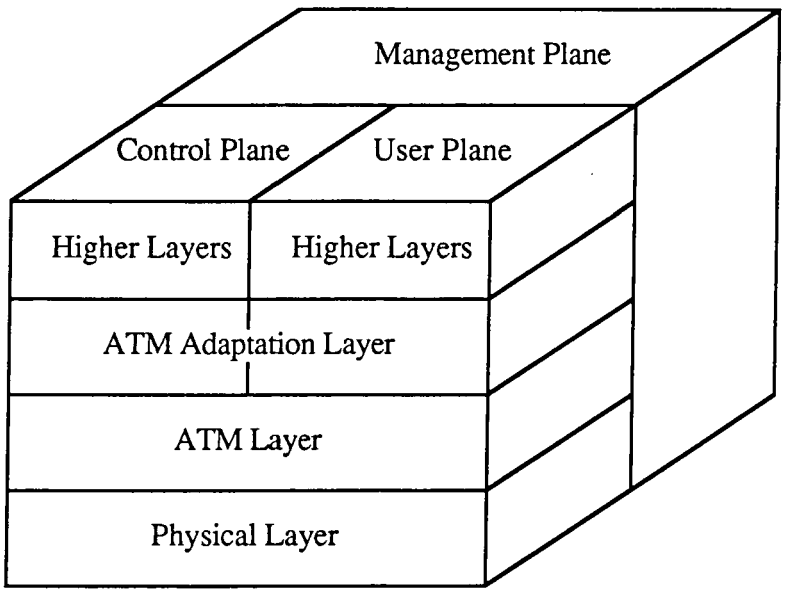

Figure 17: The layers of the A.T.M. protocol stack.

4.2.1 Cell Structure

The A.T.M. cell structure is defined in Recommendation I.361; the cell is of fixed size with a 48 octet payload field and a 5 octet header. The format of the payload remains fixed and unmodified during the whole time that the cell remains in the A.T.M. network; since error control is performed at the edges, and then only when a service requires such a feature, the field does not contain an explicit error-correcting, or -detecting, capability. The header has two formats, depending on where the cell is within the network: between user equipment and the network (U.N.I., user-network interface) provision is made for generic flow control; for communication between nodes within the network (N.N.I., network-network interface) no such provision is made and the relevant bits are made available to increase the number of virtual circuits.

The header at the U.N.I. is shown in figure 18; it contains:

G.F.C. Generic Flow control. This field is to enable the network to regulate the arrival of cells from a source external to the network. Details of the values to be used have yet to be defined. 4 bits.

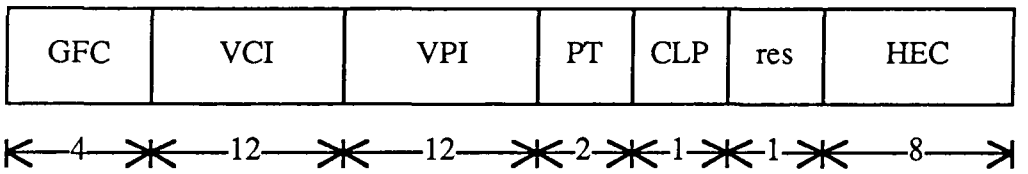

Figure 18: The cell-header format for the U.N.I.; dimensions in bits.

V.P.I./V.C.I. Virtual Path Indicator, Virtual Circuit Indicator. These fields are normally used in conjunction with each other; together they are used to route cells through the network from one edge connexion to another. The values assigned have only local significance and may change at various intermediate nodes. 8 bits V.P.I., 12 bits V.C.I..

P.T. Payload type. This field is used to describe the type of data that the cell contains; assignments of the values have yet to be made, but it might, for example, be used to distinguish signalling traffic from user data, or to indicate some form of priority. 2 bits.

C.L.P. Cell Loss Priority. A cell with this field set is more likely to be discarded at times of network overload than one with the field cleared. 1 bit.

H.E.C. Header Error Control. This field contains a CRC-8 code which is used to protect the header (only) against corruption during transmission. Whilst it is part of the A.T.M. header, it is computed and tested in the Physical Layer functions before cells are passed up to the A.T.M. layer. 8 bits.

res Reserved. A single bit field that is reserved for future use. 1 bit.

The header for use at the N.N.I. is shown in figure 19; it is identical to the header for the U.N.I. with the following exceptions:

- There is no G.F.C. field.
- The V.P.I. field is expanded to 12 bits to provide increased routeing capacity within the network.

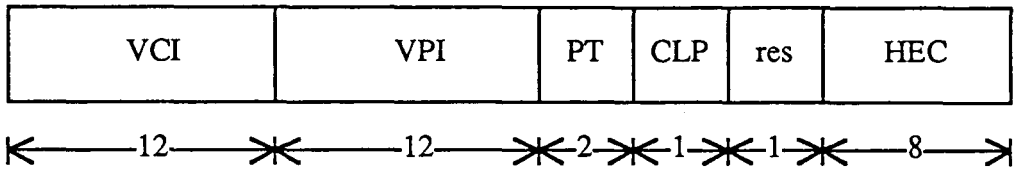

Figure 19: The cell-header format for the N.N.I.; dimensions in bits.

4.2.2 Routeing

The primary rôle of the cell header is to provide sufficient information for the cell to be routed through the network. Since the size of an international network has to be considered it is impractical to use datagram headers and destination addresses within each cell (the 20 address bits available in the header at the U.N.I. would provide for approximately 1 million possible destinations, which is insufficient to cover the needs of U.K. alone), so a virtual circuit approach has been adopted. To simplify routeing of connexions at the trunk exchange, the routeing bits have been partitioned into two parts: a virtual circuit, with each connexion having its own circuit identification, and a virtual path, which is then used to group connexions using the same route, or part of a route, through the network. At the trunk exchanges, connexions are routed solely on their virtual path identifier, but at local exchanges and elsewhere the full address part may be used: normally, at an exchange the circuit number will change to represent the value required for the local link; at trunk exchanges only the V.P.I. changes, while the V.C.I. remains constant; elsewhere, both the V.P.I. and the V.C.I. may change between input and output.

For some services, notably user-network and intra-network signalling, establishing a virtual circuit before communicating might be inefficient, or even impossible (the signal to create the circuit would need to be sent, but this requires that a circuit already exists...); in such cases permanent, or semi-permanent, circuits can be established. It is even possible that some user-user services will also work better in a connexionless environment; in this case a connexionless type service can be established by reserving a V.P.I. value and then assigning the V.C.I. values

to represent individual destinations within the network: individual communicating pairs are resolved using the A.T.M. adaptation layer. To implement this the network has to be partitioned into zones [55].

4.2.3 The A.T.M. Adaptation Layer

The A.T.M. adaptation layer is the functional layer above the A.T.M. layer; as such, it strictly lies outside the formal definition of the A.T.M. network: its functionality is service dependent and the fields that it 'defines' are only interpreted by equipment that lies beyond the edges of the network itself. However, despite the fact that implementation is not mandatory, all services will require some form of adaptation, and most are likely to fall into one of the four classes of adaptation layer framing. The layer handles such functions as: segmentation and reassembly of messages; error recovery and retransmission; handling of lost or misinserted cells; flow control; and timing control.

Four basic classes of adaptation are defined in Recommendations I.362 and I.363. The features supported are shown in table 1.

	Class A	Class B	Class C	Class D
Timing relation between source and destination	Required		Not Required	
Bit Rate	Constant	Variable		
Connexion Mode	Connexion Oriented			Connexionless

Table 1: Suggested A.T.M. Adaptation Layer classes.

4.3 Enabling Technology

The ability to build the high-speed networks required for broadband I.S.D.N. is a direct result of advances in hardware technology. Not only is greater capacity available because of the increase in the speed at which it works but, also, the probability of failure is decreasing: if the rate of failure were not to decrease then

a system running ten times as fast would be likely to fail ten times as often; in practice the situation is better than this.

4.3.1 Fibre Optics

One of the main technologies that makes a B.-I.S.D.N. possible is the use of fibre optics with semiconductor-laser technology. Copper-based communication links have always been subject to electromagnetic interference (E.M.I.), which is commonly of a bursty nature and can distort the connexion for several bit periods so that the signal is beyond recovery; single bit errors tend to be more of a secondary issue, but they can still cause problems. The non-negligible resistance and leakage of the copper connexion causes attenuation; on runs of more than a few kilometers repeaters are required to boost the signal back to its original level and to 're-shape' the pulse edges.

With fibre optics the primary source of interference, E.M.I., no-longer applies, so the probability of an uncorrectable burst of data occurring is almost zero. Attenuation within the fibre is significantly lower than for an equivalent length of copper and consequently repeaters are needed less frequently, thereby increasing the reliability of a link. In addition, mono-mode fibres have (as their name implies) only one mode by which the light signal can propagate down the fibre; this eliminates multi-path distortion and further increases the possible distance between repeaters. The current capacity of fibre-optic systems is not limited by the fibre itself, but by the speed with which the lasers and detectors can be made to operate reliably; it is to be expected that this will continue to increase in the future as the technology matures.

4.3.2 Switching Technology

To support the very large amount of traffic that a B.-I.S.D.N. will be required to carry, new types of switching technology are necessary. The concept of a circuit running through an exchange, carrying a single call from input to output,

Figure 20: 2-input, 2-output switching element that forms the basis of the Starlite switch. a) crossed over outputs, b) straight through.

was weakened with the introduction of digitized voice services: it is almost totally destroyed by the packetization of the samples; circuits no-longer exist in any physical sense, only the logical sense remains. In traditional packet-switched networks the amount of processing that was performed on each packet at each switch in the network made it feasible for the switching process to be performed in software, without the need for complex hardware to parallelize the process. In order to achieve the higher throughput of a B.-I.S.D.N. the lower layers of software were simplified to the extent that most of the remaining functions could be implemented in hardware; the switching routines would be swamped if a software process were used.

A highly parallel switching architecture is the Starlite switch [56]. It uses fixed-size packets and routes these in parallel waves across the switching architecture; it is non-blocking provided that two, or more, packets are not routed to the same destination at the same time. The basic element is a 2-input, 2-output switching device that can either pass both inputs straight through or can cross them over to the opposite outputs (figure 20): the main features of the switch are built up from replicas of this element, making the whole fabric well suited to implementation using V.L.S.I. technology). There are two main phases to the switching process (figure 21): a sort phase, which uses the elemental switches arranged as a Batcher network to perform a 'perfect shuffle', and an expansion phase, which switches the sorted packets to the correct outputs. The expansion phase is sometimes known as a Banyan Network, leading to the whole switch being known as a Batcher-Banyan Network. To cope with packets addressed to

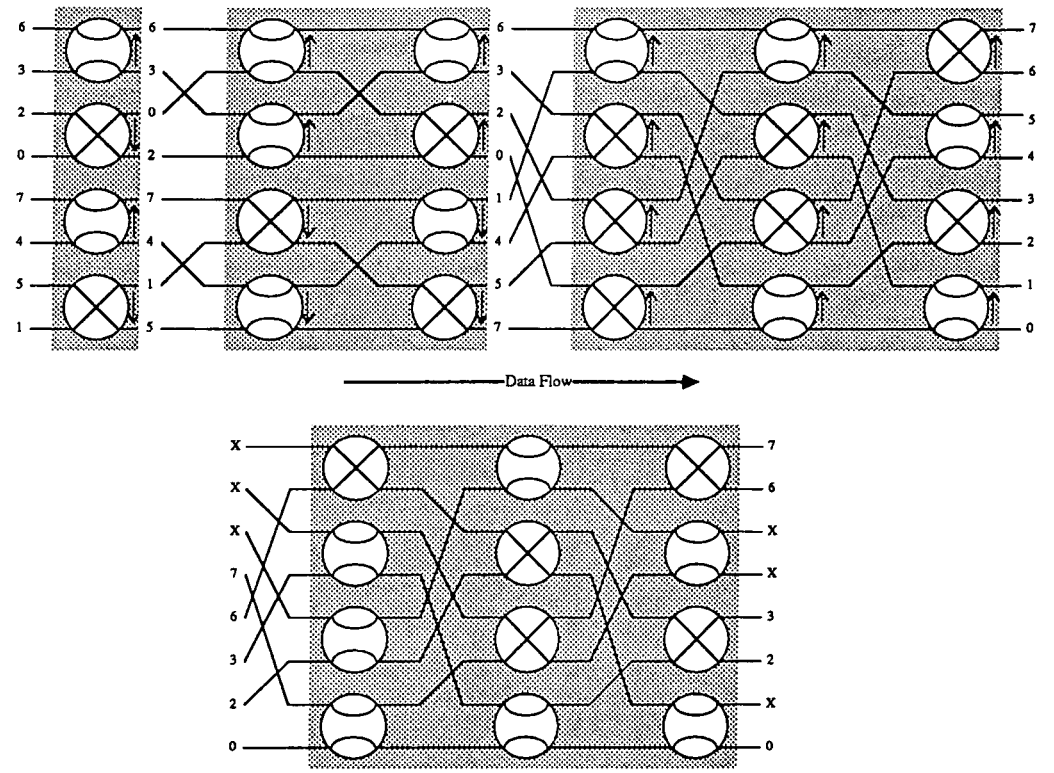

Figure 21: Connexion graphs for the two main phases of a Starlite switch. The top graph performs the perfect shuffle to sort the inputs into order (the arrows point to the output that should have the larger of the two inputs); the lower graph is the expansion phase that uses the bit fields of the output port to determine the route.

the same output a trap phase has to be used between the sorter and the expander (figure 22); this re-routes packets, that would otherwise be lost, back to the input of the sorter for processing on the following pass; the loss rate of the exchange can be determined by the amount of capacity within the trap that can be re-routed in such a manner. The concentrators are used to keep the number of switching elements as low as possible: the sort phase requires $n(\log_2 n)(\log_2 n + 1)/2$ basic elements for n inputs; the expander requires $m(\log_2 m)/2$ basic elements for its m inputs: for large switches the probability that all inputs would be active at once is very small and large economies can be made by using concentrators to reduce n.

Another possible switch architecture that has been proposed is the Orwell

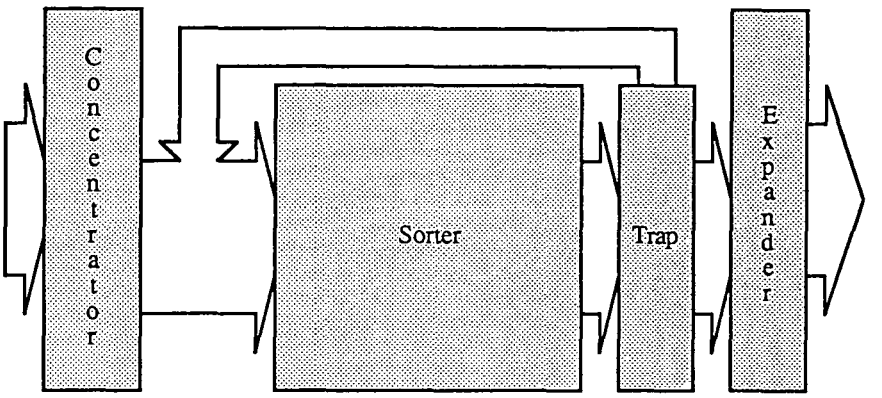

Figure 22: Stages of a Starlite switch. The concentrator phase is used to reduce the number of inputs to the sorter; the trap phase to catch two or more cells that are routed to the same output which would get blocked in the expander; all but one are recycled back to the input for switching on a subsequent pass.

Torus [57]; this would be used mainly in low capacity exchanges where its greater flexibility would be more important than very high throughput. The protocol uses multiple slotted rings, running in parallel, and novel access control mechanisms, to bound access delays for time-critical services. The protocol is discussed in more detail in the next chapter.

4.4 Traffic and Services

In order to dimension correctly the capacities of various elements within a B.-I.S.D.N. the types and expected loads of services offered need to be analysed in detail; provision for future growth of the services also needs to be included; this is likely to prove extremely difficult to calculate since the services requiring most bandwidth are, as yet, untried, and their likely penetration into the market are, at present, unknown. It is the necessity to reduce the risk of forecasting errors that makes the requirement for a single transmission network so important; errors in under-forecasting for growth in one service may well be compensated by over-forecasting in another.

4.4.1 Connexion Control

If all the services carried by a B.-I.S.D.N. used constant bit-rate encoding then calculating whether a service could be safely carried without affecting the delays and causing overloads would be a relatively simple extension of the techniques used for traditional voice networks (the main difference being that services could require differing amounts of bandwidth and that these would not be integral multiples of some base rate); in such circumstances connexion control is not particularly difficult. However, some services, such as interactive computer-data transfer, are naturally bursty in nature and others, such as compressed video signals, often require a variable bit-rate coding scheme; when several of these services are carried simultaneously on a single network then some network capacity can be saved by understanding the statistical processes involved and working out the likelihood of an overload occurring.

There are three statistical parameters to a variable bit-rate coding scheme that are of interest to network operators: mean, standard deviation (or variance) and peak bit rates. It is, at present, unclear which of the latter two is the most important in terms of dimensioning the network: one approach suggested for connexion control is for the call to be accepted purely on the basis of its peak bandwidth requirement; while this would enable cell loss due to overload to be completely avoided, significant benefits available from statistical multiplexing would be lost, making the network inefficient. When connexion acceptance is based on statistical calculations then the 'acceptance surface' (a multi-dimensional surface indicating the load limits for various combinations of traffic) is no-longer a plane, but becomes distorted due to the differing standard deviations: attempts have been made to calculate this surface in advance [58], and others have suggested that the surface might be determined when the network is in operation (for example, by using neural networks [59]).

4.4.2 Charging and Policing

Two areas that are still a major subject of debate are charging customers for use of the network and policing connexions, particularly V.B.R. connexions, to ensure that they do not try to transmit more than had been originally stated. The two areas are similar in that the same techniques and problems apply to both, although different techniques may be applied. The main options available are to monitor the mean rate or the peak rate, or possibly some function of both: charging only for the peak bandwidth used is, however, difficult to justify unless the bandwidth is guaranteed; conversely, charging for the mean bandwidth used does not reflect the spare capacity that has to be set aside to cope with the statistical fluctuations. Policing is important, particularly when statistical connexion acceptance is performed, to ensure that the traffic load presented to the network matches that which was negotiated when the call was initiated. The problems in both cases are concerned with the shear bulk of information that has to be processed; a simple network may well be carrying several million cells every second, and to monitor each one presents an insurmountable burden for the policing and charging functions.

Chapter 5

Orwell Model Simplifications for Network Level Simulation

Rings form one of the three basic types of L.A.N. topology (the others being star and bus) and three basic types of protocol have been developed for use with them. By far the most popular of these are token based protocols, whereby the node holding a token is given exclusive access to the ring. Register insertion is another alternative; messages can be inserted onto the ring, delaying any existing traffic by passing it through a shift register. The third, but nowadays less favoured, approach is to use a slotted ring protocol: the ring is divided into slots which circulate around the ring; a node wishing to transmit a message waits until an unfilled slot is found, changes the header and transmits the message in the body of the slot.

Slotted ring protocols were unpopular for several reasons: a monitor node is required to ensure that slots that become corrupted can be identified and regenerated (correct behaviour of the ring is critically dependent on correct behaviour of the monitor); to get a reasonable number of slots onto the ring delays have to be inserted at each node and one node, normally the monitor, has to be able to adjust its delay so that there are an integral number of slots; and the efficiency of slotted rings is generally poor since the ratio of header to body is normally high. Its greatest advantage over token-based protocols, however, is that more that one node can be transmitting information at a time, using different slots on the ring. Acknowledgement of delivery is normally made by releasing the slot at

the source (correct receipt there is taken to imply correct delivery at the destination); the node may not refill a slot that it has just released, ensuring that the slot is passed to the next node and thereby ensures fair access to all nodes on the ring. A typical implementation of a slotted ring is the Cambridge Ring protocol (British Standard BS6531).

Examination of existing protocols has indicated that those based on a slotted ring are probably the best suited for carrying delay-sensitive speech, but simulation studies of high-bandwidth Cambridge Rings have indicated that there are still significant limitations when operated under high load [60] and, further, load control is difficult since there is no relevant parameter that can easily be extracted from the ring. The Orwell protocol was developed after making a detailed study of the limitations of the Cambridge Ring protocol: it was found that by introducing destination release of slots, and by adding a novel, distributed, load control mechanism to bound access delays, a viable level of performance could be obtained [61, 62]. For higher capacity networks multiple, synchronized, rings can be used and such a network is known as an Orwell Torus.

Whilst detailed simulations of a single ring have been made, under a variety of load and traffic services, there has, as yet, been very little investigation made into the behaviour of an Orwell torus, or ring behaviour in multi-ring systems. The reason for this, at least in part, is because of the large amount of simulation time required to investigate networks of Orwell rings: a single simulation run of one ring takes, typically, a couple of hours on a VAX, or three times as long on a Sun 3/50 work-station for just a couple of seconds of simulated time.

There are three main options available to try and reduce the amount of time required for simulation. The first and, almost certainly, least feasible option is to use a larger and faster conventional computer than a VAX; this may reduce the amount of C.P.U. time required, but it is unlikely to decrease the total time for one simulation because of the higher demand placed on such machines. The second option is to break down the simulation model into processes that occur

concurrently and to redesign the model to take advantage of parallel processing architectures such as the transputer; this option looks promising, despite the fact that an individual processing element will have less power than some single C.P.U. machines, because the total power can be increased by simply using more processors. The third option is to create a new model that has the same external functionality as (or as close as possible to) the original model, but to make simplifications internally in order to reduce the computational requirements: if successful this third option can either be used on its own, with the original computer, or with either of the other options to reduce simulation time still further. This chapter considers various simplifications of the model of the Orwell protocol that were investigated while attempting to reduce the amount of computation required during simulation. All the results included here are based on simulations using the Orwell simulator [63, 64], written in Simula '67, and on modifications made to that program.

5.1 Overview of the Orwell Protocol

The full specification of the Orwell protocol, detailing its running actions, start-up procedures and details for ensuring slot integrity is available in the specification document [57], but an overview of the running actions is given below for completeness.

5.1.1 Ring Actions

An Orwell ring consists of a series of nodes connected by a closed communications loop, figure 23. A number of slots circulate around the loop in a single direction. Each of the slots may be in one of three states: full, empty (known as trial) or reset; these states are explained below.

Each node maintains a counter, known as a *d-counter*, whose initial value is an indication of the traffic that the node has agreed to carry. Each time a cell arrives at a node it is placed in an input queue; when a node finds an empty slot

then, provided the d-counter is greater than zero, it places the first cell in the slot and decrements the d-counter by one: if the d-counter is already zero then the node is barred from using the slot and must leave it empty for subsequent nodes; in this way 'hogging' of the ring is prevented.

When a node finds a full slot addressed to itself it removes the cell from the slot and marks it as empty but with its own address (it is barred from immediately refilling the slot). If a slot makes a full revolution of the ring without being seized by another node, the ring is declared to be idle and the slot is converted into a reset slot (for this reason the empty slot is usually referred to as a trial slot). A node seeing a reset slot restores the d-counter to its original level; the reset slot is passed on to each node until the whole ring has been reset.

In this way, the ring can undergo a reset for either of two reasons, although the single method is used to detect both: either all the nodes on the ring have become idle and have no traffic for the ring, or because they are blocked from accessing the ring because their d-counter has reached zero. In either case the ring will rapidly approach a time at which all of the nodes are either idle or blocked; a reset then occurs and the whole process is repeated.

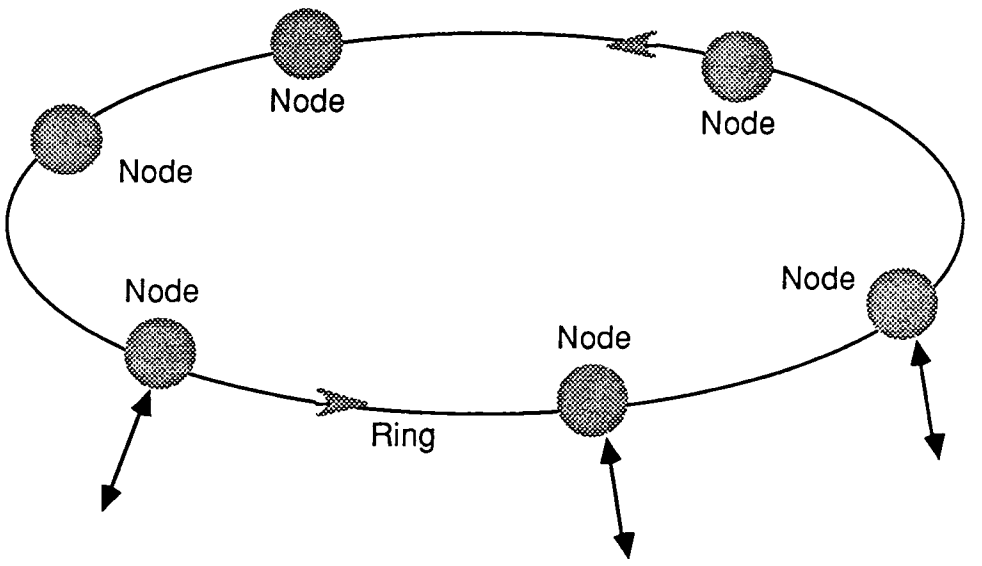

Figure 23: A simple Orwell ring

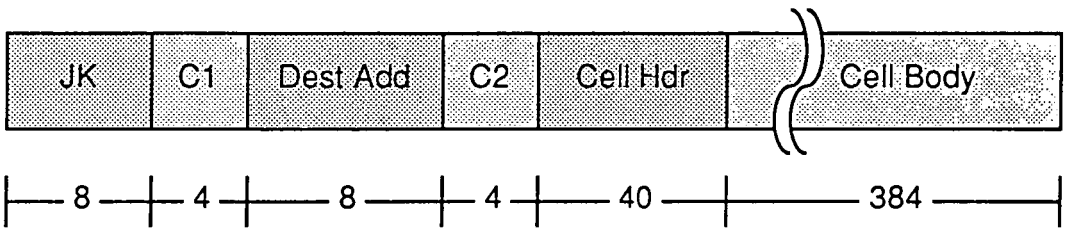

Figure 24: Slot format (field sizes in bits)

When a new call requests use of the ring, the node makes a decision, based on the current rate at which resets are occurring, as to whether carrying the new call is likely to reduce the reset rate below an acceptable minimum. If this is likely to happen the call is blocked, otherwise the call is accepted and the original value for the d-counter is adjusted accordingly.

5.1.2 Slot Format

When carried on an Orwell ring a prefix to the cell has to be added, the complete entity then being known as a *slot*, figure 24. The JK field has a unique format to guarantee synchronization at the nodes. The C1 field is further subdivided into four fields, the first two of which are used to define the type of slot; the third bit, called the monitor bit is used to prevent corrupted cells from clogging up the ring; and the fourth bit is called a broadcast bit, when set the cell will be copied by more than one node as it passes around the ring. The C2 field is mainly concerned with error protection on the slot header, and with other control and signalling functions; its behaviour is not important within the context of the work covered here.

5.1.3 The Orwell Torus

To enable Orwell rings to carry very large volumes of traffic the protocol has been designed to allow a number of rings to be able to operate together, in parallel, and in a synchronous manner: such a network is known as an Orwell Torus. Figure 25

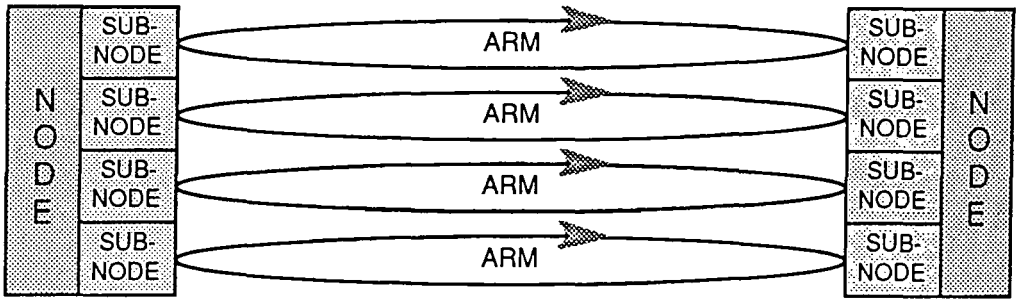

Figure 25: Torus of Orwell rings

shows an example of the torus, each individual loop of glass fibre between the nodes is known as an Arm. The slots on individual arms are staggered so that ordering is always preserved between slots on different arms of the torus (there are fairly strict limits in the different fibre lengths that can be used on each arm). All of the rings operate using a single *d*-counter, and a cell awaiting access to the torus is placed in the first slot to become available; resets operate similarly, a reset on one ring causing all the rings to be reset [65, 66].

Another advantage of the torus is the increase in reliability due to replication in the network: if a single ring or sub-node fails then the system can simply carry on operating at a reduced capacity; careful isolation between the node controller and the sub-nodes can ensure that, should a node controller fail, the sub-nodes can become transparent repeaters that take no further action on the torus other than to forward slots.

5.1.4 Calculation of the d-value

In practice, traffic on an Orwell ring is divided into three classes, each at a different priority level; in this way, traffic which is highly delay sensitive (for example, voice and signalling traffic) can be given a higher priority. To ensure that all classes of traffic still have some access to the network, the *d*-counter is also divided into three counters, each representing one of the priority levels. When a cell arrives at a node and is waiting for access to the ring it is placed

in the queue appropriate to its priority: when an empty slot is received the cell in the highest priority queue that still has an unused d-allocation is selected and the appropriate counter adjusted downwards; since the node as a whole is only barred from further access when all of the queues are either idle or blocked all classes of traffic are guaranteed some access to the ring during each reset interval, but delay sensitive services always get the highest priority.

In order to bound the delay at a ring within acceptable limits the amount of traffic carried has to be carefully controlled. There is no need, however, for a centralized call-control mechanism since the total load being carried by the ring can be determined from the reset rate provided that the ring has time to reach equilibrium between call attempts: calls are only accepted if the reset rate is sufficiently low to guarantee sufficient capacity for that call. For each new call the d-allocation is adjusted accordingly; there are several methods available for determining what value this should be and two possible methods are given here. The static allocation scheme bases the calculation on the arrival rate of cells for that type of call, A, and the maximum permissible interval between resets (Maximum Reset Interval, M.R.I.): for each call,

$$\delta d = \frac{A}{\text{M.R.I.}}, \tag{16}$$

and the d-allocation for a single priority is the sum of all the appropriate δd's rounded to the next largest integer; for V.B.R. calls, A is not necessarily the mean cell arrival rate, but may be slightly higher to allow for statistical variation. In the dynamic allocation scheme the d-allocation is adjusted based upon whether it was fully used over the preceding reset intervals: if the full d-allocation were used over, say, the preceding three intervals then the allocation is increased by one; if it were not fully used in each of the intervals then its value is decreased by one. This is subject to a maximum which is based on the capacity of the ring: the allocation for the lowest priority queue can either be a fixed constant or be adjusted to represent the difference between the maximum for the ring and the amount claimed by the other queues.

Computer-data traffic, which is usually the service with the greatest delay tolerance is normally allocated to the third queue which commonly has a permanent d-allocation of 1 or 2. This ensures that while the reset rate is high a large proportion of the ring bandwidth is available for such services, but as the reset rate drops (i.e. occur less often) then such services are 'throttled back' and priority given to those that are delay sensitive; some bandwidth, however, is always guaranteed.

5.2 First model

5.2.1 Algorithm

This model emulates the behaviour of the ring by using an array filled with random node numbers to represent the searching action of slots. The algorithm is reproduced below.

Each node, i, on an Orwell ring has an amount of bandwidth allocated to it that is stored in its 'd-counter', d_i; d_i being proportional to the number of calls being carried. Then, assuming that there are N nodes on the ring, let

$$S = \sum_{i=1}^{N} d_i. \tag{17}$$

An array, Q, of size S is then filled using the following algorithm:

```
for each node, i
        repeat d_i times
                j := random number between 1 and S
                if Q_j is filled then
                        increment j until Q_j is unfilled
                fi
                Q_j := i
        end
end
```

Once the array has been filled, it is scanned in order using the following algorithm. This simulates the random manner in which the slots are accessed by the nodes waiting on the loop. A complete pass of the array with no cells switched represents a trial slot traversing the entire loop without being claimed and a reset occurring.

$j := 1$
repeat
    if Q_j filled **then**
        if cell waiting on node Q_j **then**
            switch cell
            $Q_j :=$ empty
        fi
    fi
    $j := (j \bmod S) + 1$
until All Q_x are empty **or** one pass of j with no cells switched

Since, on average, a slot is filled by a cell for one half of one ring rotation, and because the slot cannot be filled again until the slot has reached the node after the one at which it was released, then the slot is in use for, on average, $1 + N/2$ nodes and the proportion of each ring rotation for which the slot is in use is

$$\frac{1 + N/2}{N}. \tag{18}$$

If there are K slots on each ring in the torus, and R rings, there will be a total of KR slots. If the slot rotation time is t seconds, then the number of cells that are carried in one second is

$$\frac{1}{\tau} = \frac{NKR}{t(1 + N/2)} \tag{19}$$

giving τ as the mean time between each cell being switched.

To take account of the fact that the first time the ring is found to be idle would probably not cause a reset to occur, the algorithm was implemented in a slightly modified manner to permit this feature to be incorporated: the searching

algorithm was augmented with a status counter that was reset to zero each time a reset occurred; the ring was not reset until the status counter had incremented to a pre-calculated limit (this calculation being based on the minimum time that a real ring takes to reset when completely idle, i.e. one slot rotation time plus the time required to get to following node). Several algorithms were used for determining how the status counter should be incremented. The first was to reset the status counter to zero each time a cell was carried; the second to allow the counter to increment to a certain value each time a cell was switched, and then to pause it at this value until the ring became idle before letting it increment up to the limit; the third was simply to allow the status counter to increment up to a fixed distance from the limit and pause it at this level until the ring became idle.

Initially the delay between switching each cell was maintained as the constant, τ.

5.2.2 Results

Simulations were performed on an eight node network connected by a 140Mbit/s ring. Only voice traffic was offered to the ring, and the auto-reset mechanism within Orwell was disabled. The cells used were of a different size to the now adopted values of 45 octets body and 5 octets header, at 16 octets body and 5 octets header: these values were maintained throughout this series of simulations so that comparisons could be made. The mean call holding time was set to a tenth of a second.

From these values the theoretical capacity of the ring can be calculated. The usable bandwidth, B' (efficiency) of the ring is given by

$$B' = \frac{\ell_c}{\ell_s} B, \tag{20}$$

where, B is the bandwidth of the ring, ℓ_c is the size of the cell and ℓ_s is the size of the slot. Since each cell uses a slot for an average of $(1 + \frac{N}{2})/N$ of a rotation, then the carrying bandwidth, B'', is given by

$$B'' = \frac{B'}{(1 + \frac{N}{2})/N}. \tag{21}$$

Since each voice call requires a bandwidth, B_v, of 64 Kbits/s (the simulator only generates traffic in one direction), then the call capacity, C of the ring is

$$\begin{aligned} C &= \frac{B''}{B_v} \\ &= \frac{\ell_c}{\ell_s} \cdot \frac{N}{1+N/2} \cdot \frac{B}{B_v}. \end{aligned} \tag{22}$$

This value is an upper bound on the carrying capacity of the ring, and it ignores the reduction in available bandwidth caused by the trial and reset slots.

For the ring simulated in these experiments, therefore, the maximum traffic capacity of the ring is equivalent to 2,666 calls. This is an absolute maximum for the ring; in practice the load control mechanism would limit the number of calls carried to somewhat less than this in order to hold the queueing delay within acceptable bounds.

Initial runs on the algorithm were done over a simulated time of 0.1 seconds after a warm-up period of 0.1 seconds; whilst these times are very short, and it is clear that the ring has not been given time to reach equilibrium, it is the relative performance of the simplified model when compared with the full model of the protocol that is of interest. The first set of simulations were performed using the 'backing off' technique for the status of the loop, holding the status counter at two less than the maximum value; intuitively this can be justified in that as a loaded ring approaches a reset, there are some empty slots circulating around the ring, whilst some slots are still carrying data. Figure 26 shows the mean reset interval as a function of carried load; it is clear from these graphs that the model has a significantly lower mean, enabling it to accept a much higher number of calls than the full protocol. Figure 27 shows the mean of the queue lengths as a function of carried load, and again it is clear that the amount of queueing caused by the model is significantly lower (the queues in Orwell are approximately a factor of ten longer).

The 'backing off' of the status counter appeared to be causing the ring to reset more rapidly than was desired, so some simulations were performed at the other extreme, i.e. the reset status counter was returned to zero after each cell

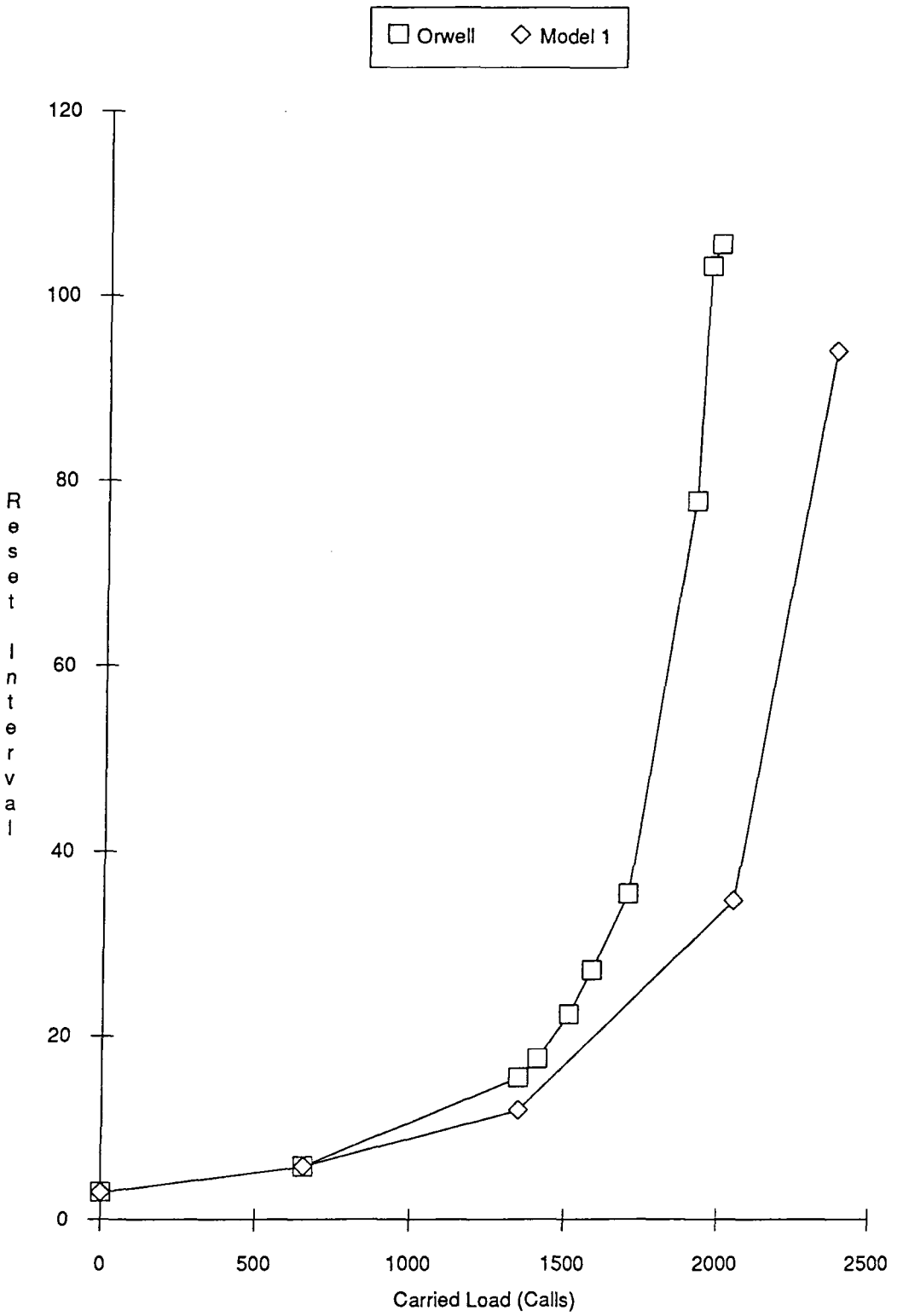

Figure 26: Graph showing the mean reset interval (in μs) against carried load for Orwell and the first model using 'backed off' resets

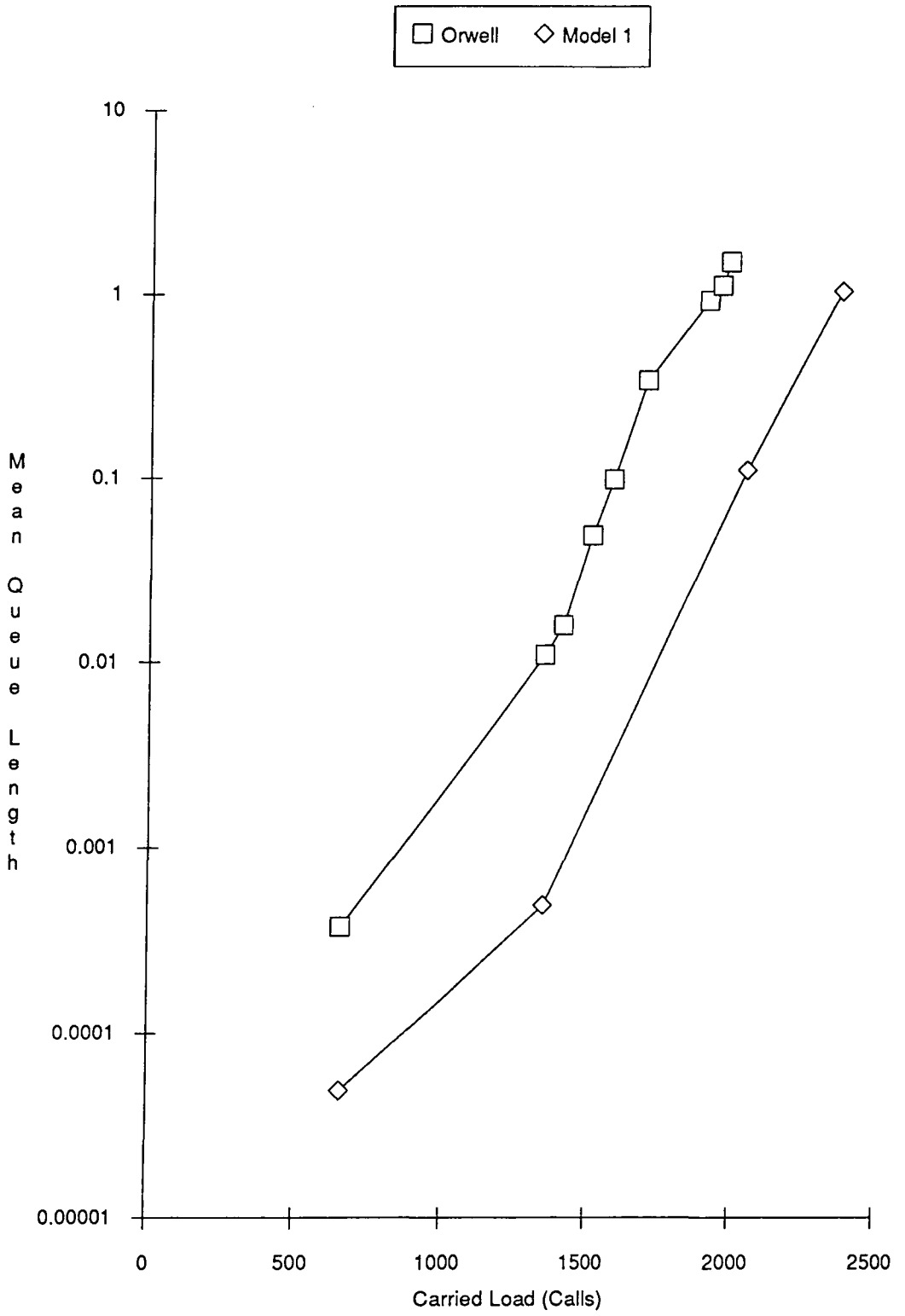

Figure 27: Graph showing the mean queue length against carried load for Orwell and the first model using 'backed off' resets

that was switched; in this way the ring only resets after a prolonged period of idleness. Figures 28 and 29 show the reset behaviour and queueing behaviour of this variant. They show that for low and medium loads, the reset interval is larger than that of Orwell, while for high loads the ring is still resetting too rapidly. The queueing can be seen to be significantly closer than for the 'backed off' model, but the queues are only of identical length at the point where the reset behaviour is least accurate.

In an attempt to match the queue lengths more accurately, it was decided to introduce a random element into the delay between switching cells, the justification being taken from the fact that an M/D/1 queueing system has a mean queue length half that of an M/M/1 system. Obviously the service time on Orwell cannot be a negative exponential, since the cell is of fixed size and the propagation delay around the ring (part of the service time in this model) has an upper bound of one rotation delay. However, as a first approximation to the service characteristic, a negative exponential service time was used, since this should form an upper bound on the degree of randomness of the service time.

The results of runs using the exponential service characteristic are shown in figures 30 and 31. It can be seen that the effect is to reduce the reset interval slightly at all loads, but has only affected the queueing at low loads; at high loads the amount of queueing is unaffected.

The results correlation obtained thus far, was fairly poor, particularly when it is considered that the reset interval determines the maximum load that the ring can accept. In addition to this, the model was taking significantly longer to execute than simulations of the full protocol, and since the carried loads at certain offered loads were similar this could only be explained as the result of using a poor algorithm. It was realized that the array filling and searching was very inefficient; in particular, at low loads a large array was being filled with random numbers and then not used because the ring was idle. In addition, the searching algorithm for the array was inefficient: to discover that the ring was in fact idle, the algorithm would have to check all the locations in the search

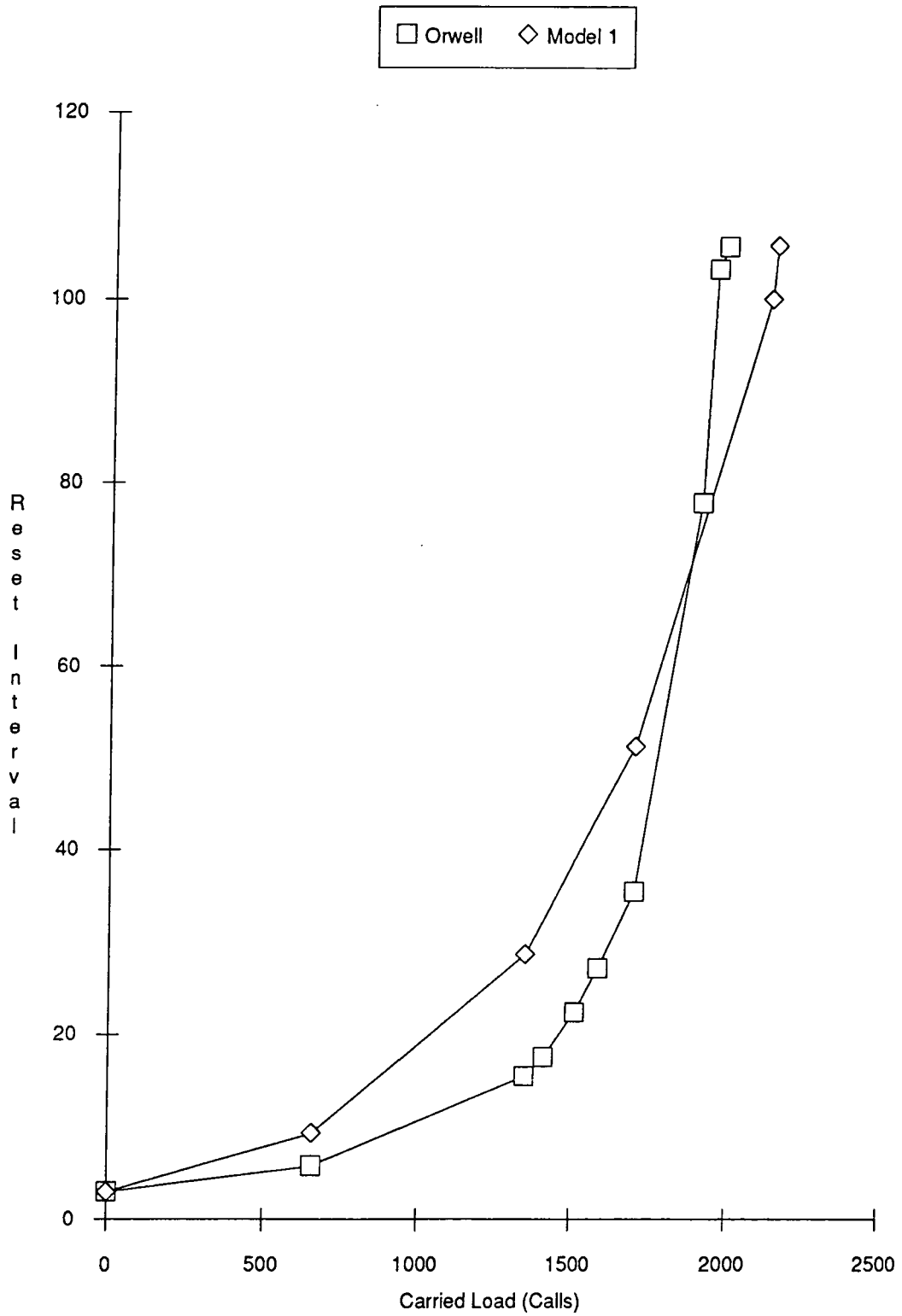

Figure 28: Graph showing the mean reset interval (in μs) against carried load for Orwell and the first model using 'totally idle' resets

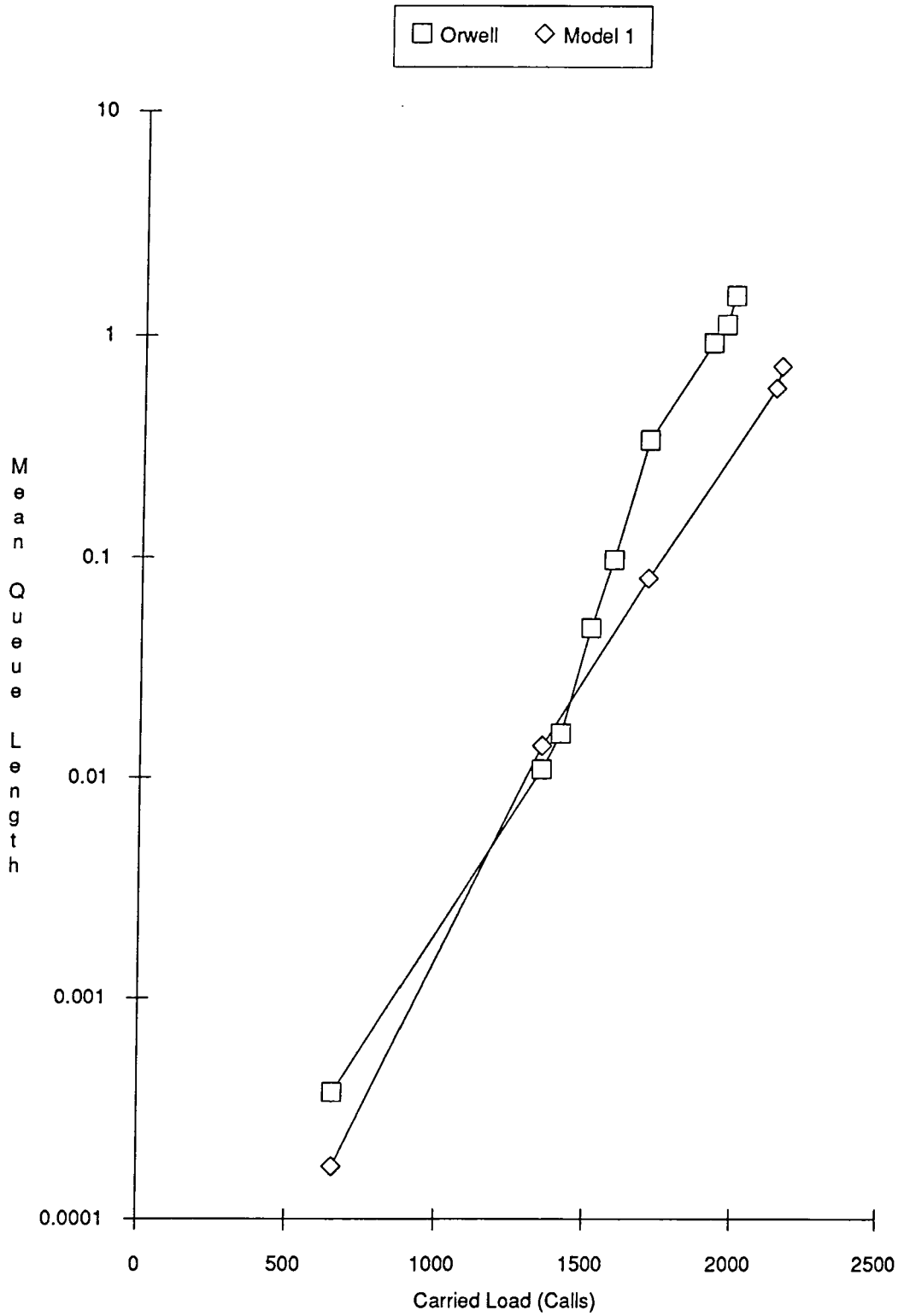

Figure 29: Graph showing the mean queue length against carried load for Orwell and the first model using 'totally idle' resets

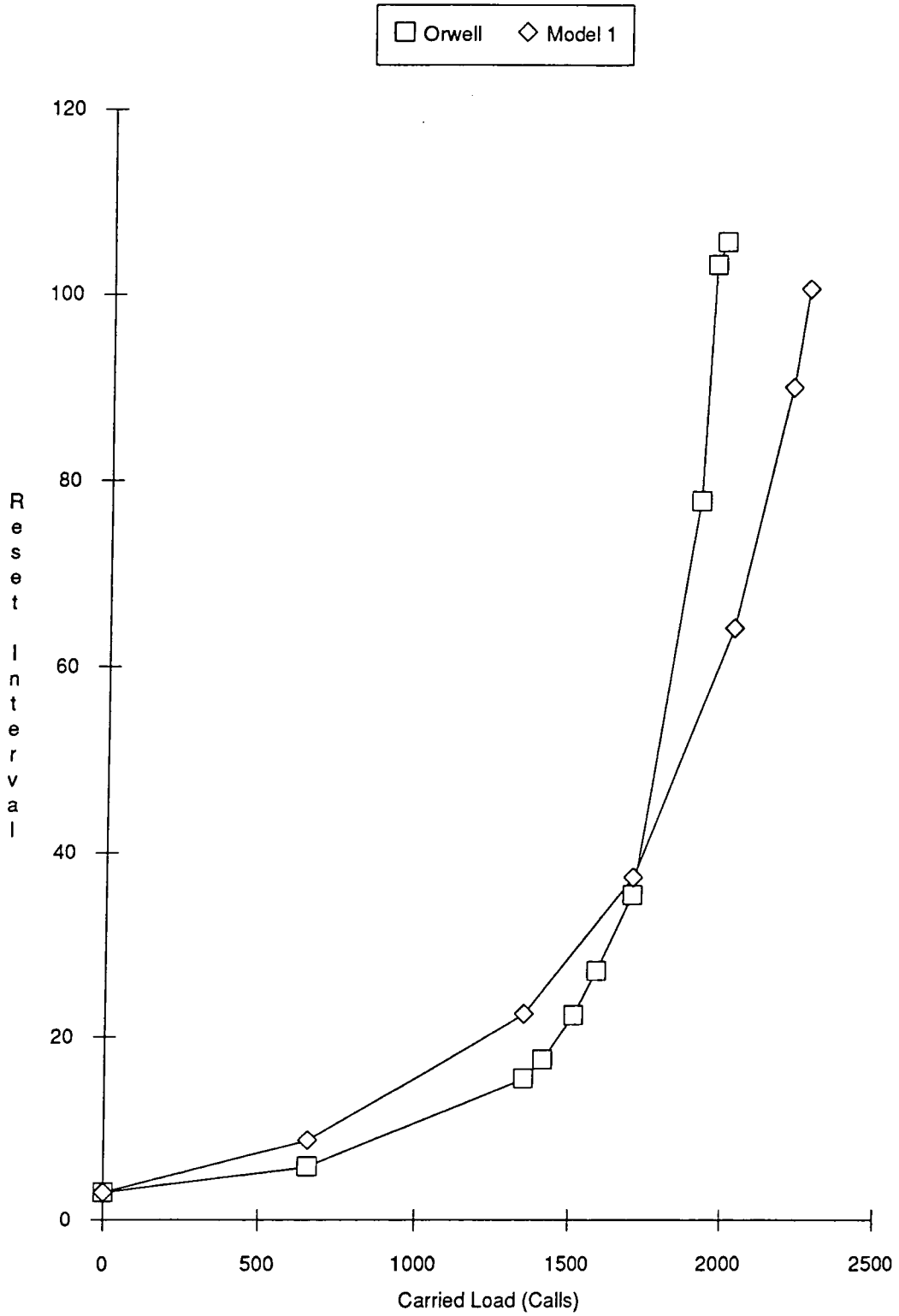

Figure 30: Graph showing the mean reset interval (in μs) against carried load for Orwell and the first model with negative exponential service time and 'totally idle' resets

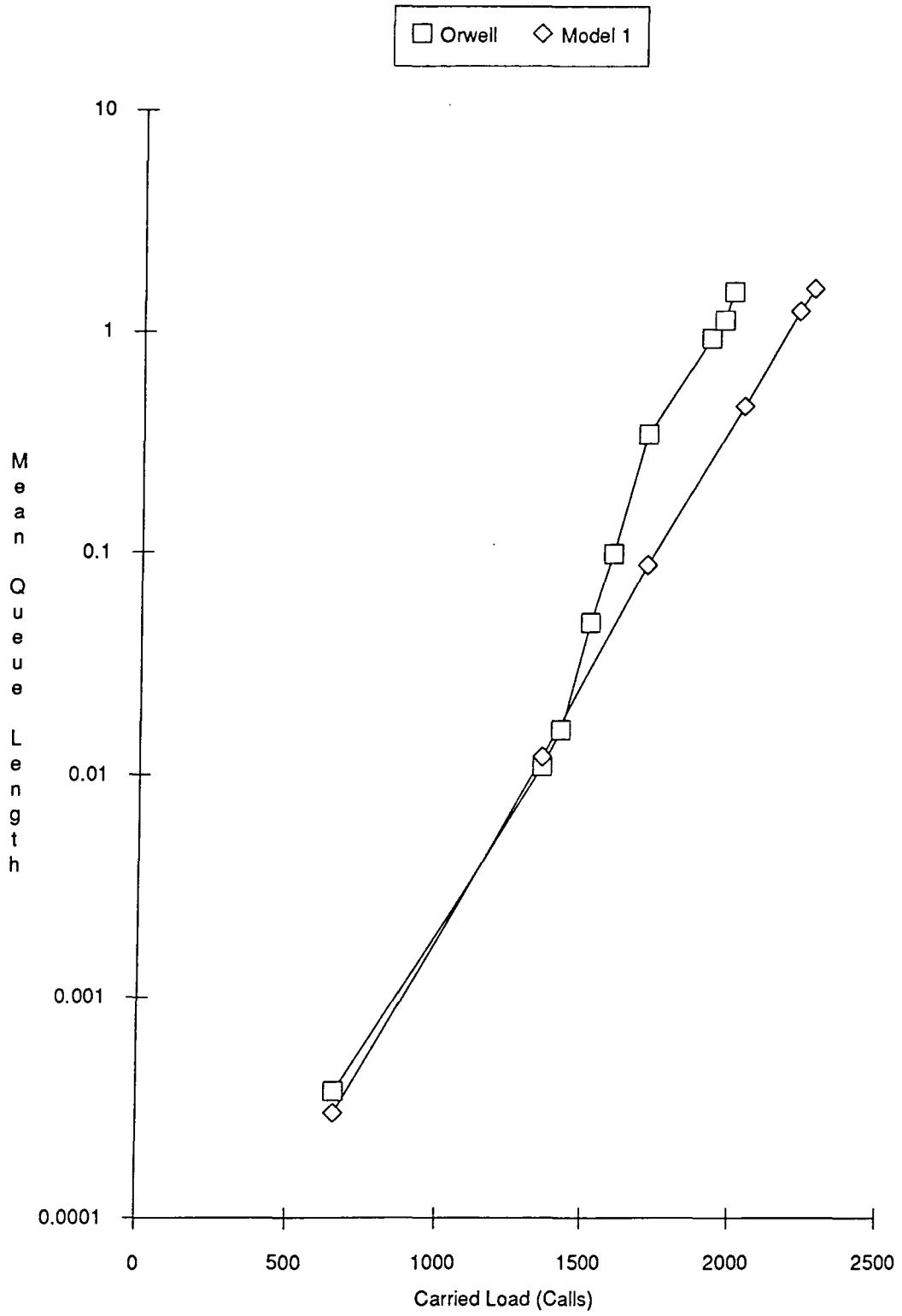

Figure 31: Graph showing the mean queue length against carried load for Orwell and the first model with negative exponential service time and 'totally idle' resets

array, regardless of whether the indicated node had already been checked. A new algorithm was developed in an attempt to rectify these problems.

5.3 Second Model

5.3.1 Algorithm

Since the first algorithm had been proved to be inefficient partly due to generating too many random numbers, an approach that avoided the redundant generation of random numbers was required, in addition it was necessary to avoid checking a node several times when trying to decide whether it was idle.

Both of these problems were avoided by using the algorithm below; in addition the overhead at each reset is reduced to that of resetting the node itself.

```
reset ring
while ring is not idle
        generate a random node number
        while (node is paused or idle) and there are unchecked nodes
                check the next node
        endwhile
        if we have a cell to switch
                switch the cell
                next status value
        else
                next status value
        fi
        wait for delay
endwhile
```

The 'next status value' is calculated by one of the methods mentioned in the first model. Switching the cell now also involves updating the d-counter at the node, a process that was not needed in the first model.

5.3.2 Results

The results for the above algorithm, using simulation runs of 1.0 seconds after 0.5 seconds warm-up are shown in figures 32 and 33. These results indicate that the behaviour of the second model is almost identical to that of the first, i.e. the ring was accepting a far greater load than the Orwell protocol. This discrepancy can be explained by inspecting the number of cells switched as a function of the size of the reset interval, figures 34 and 35. From these graphs it becomes clear that the service time of Orwell cannot be a constant, but must be a function of the offered load: an interesting, and advantageous, feature of the protocol is that this function is such that the service rate for the ring *increases* as the load increases; this should be compared with a C.S.M.A./C.D. type protocol (for example, Ethernet) where the opposite occurs, leading the network to become less efficient at high loads.

Having noted that the service rate of the ring is not a constant, the explanation is readily apparent: in Orwell the slots circulating around the ring have two phases. In the first, the slot is carrying a cell and the distribution of this phase is as noted before. In the second phase, an empty slot is *searching* for a load, and it is the distribution of this phase that is not a constant, but a function of the number of active nodes. Therefore, if accurate behaviour is to be obtained, the model needs to be modified to take this search period into account.

5.4 Third Model

5.4.1 Algorithm

The previous two models have both been characterized by having a fixed average for the service time on the ring. In order to enable a load dependent service time to be implemented a couple of changes had to be made to the simulation program. These changes entailed keeping a record of the number of nodes that had cells ready for switching (and that were not in the paused state); alterations

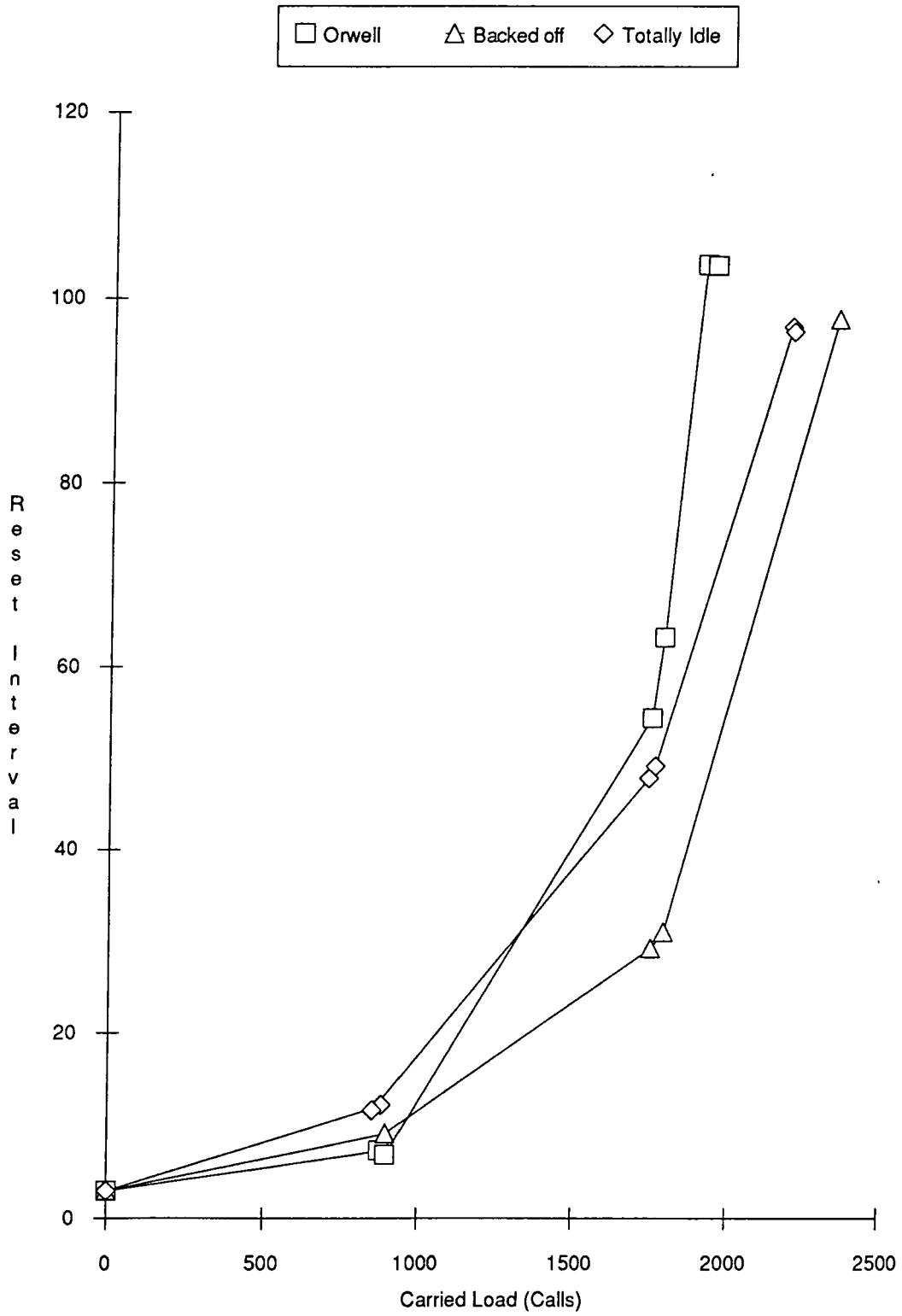

Figure 32: Graph showing the mean reset interval (in μs) against carried load for Orwell and the second model

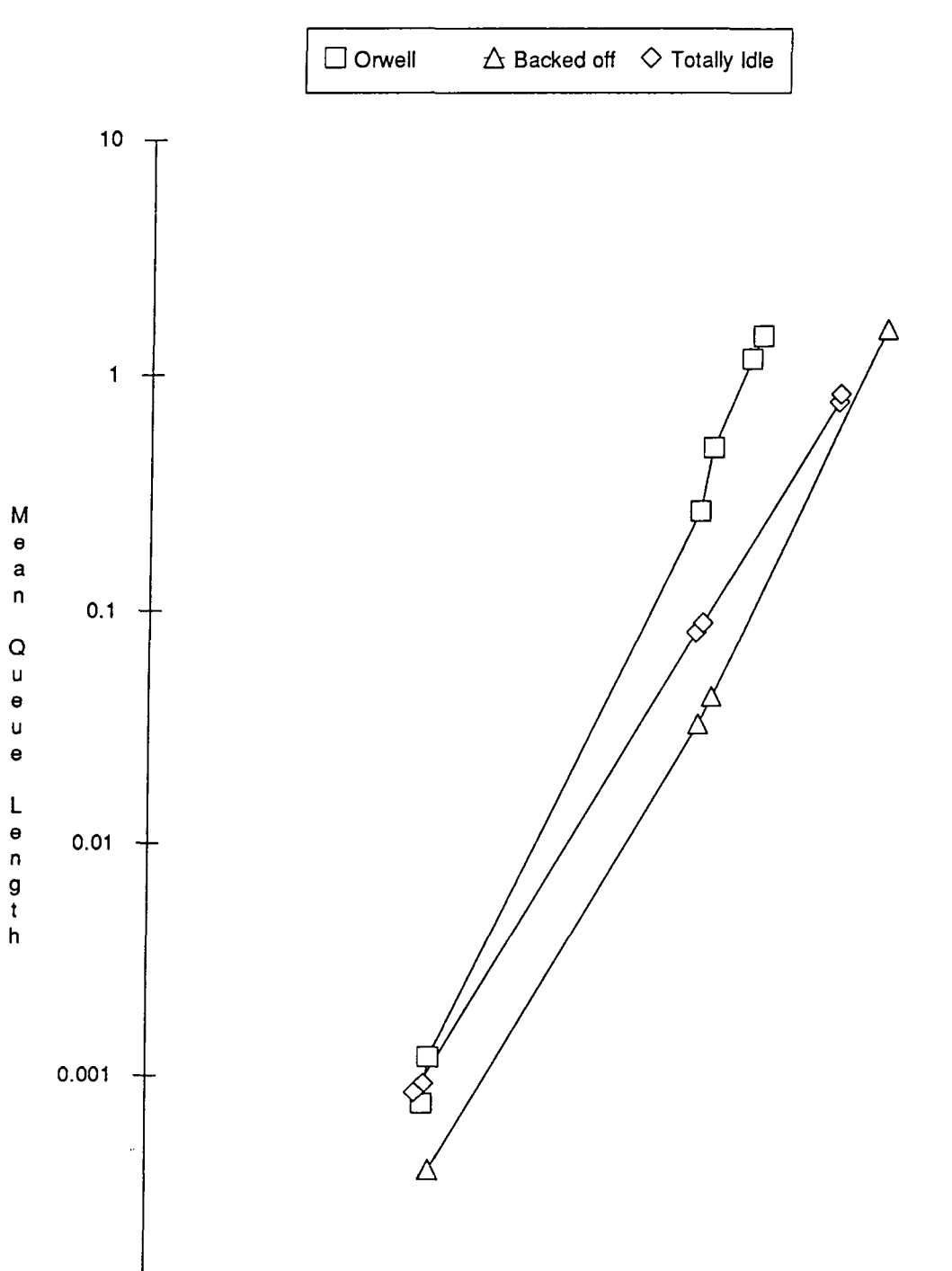

Figure 33: Graph showing the mean queue length against carried load for Orwell and the second model

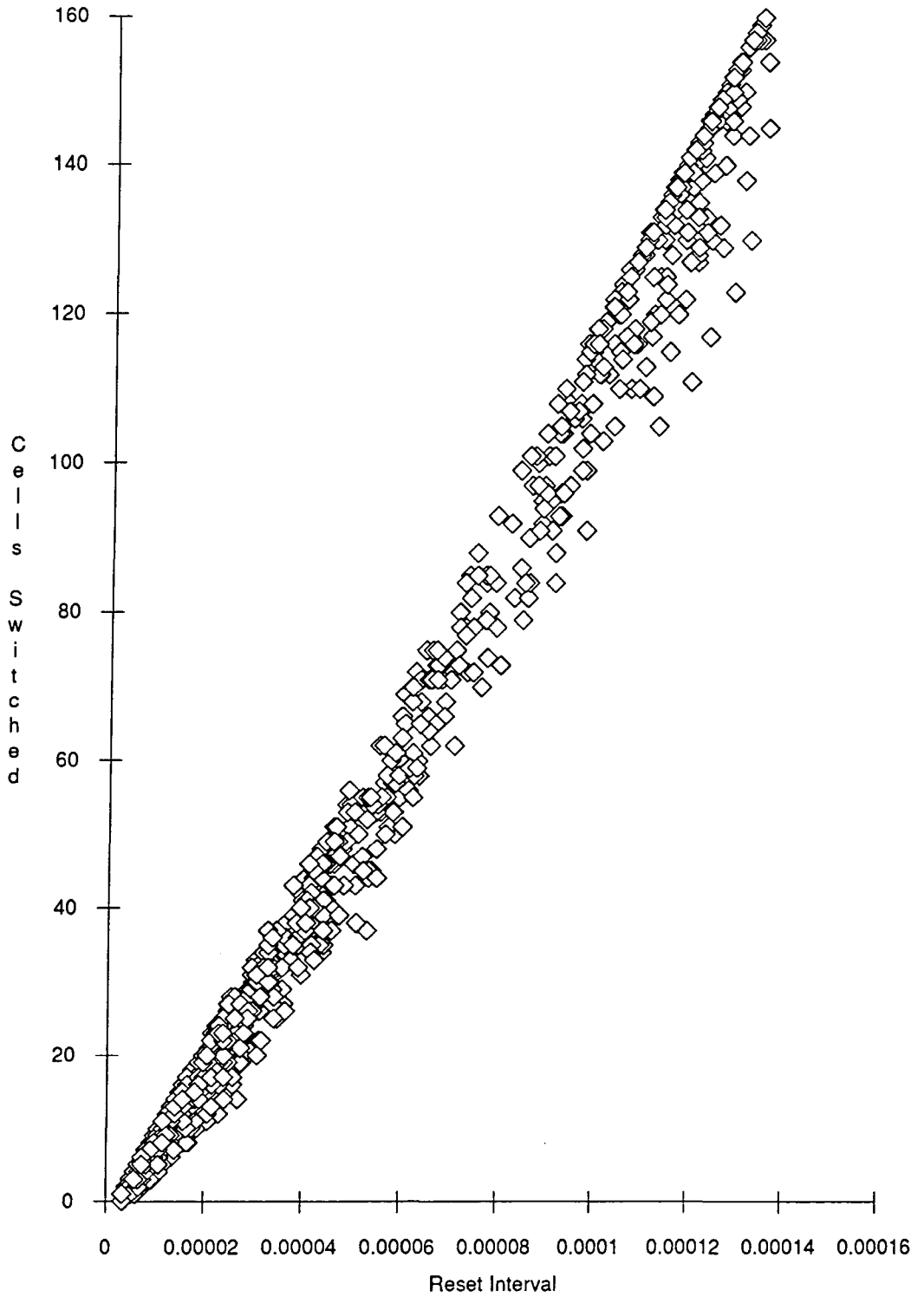

Figure 34: Graph showing the number of cells switched as a function of the size of reset interval for the model

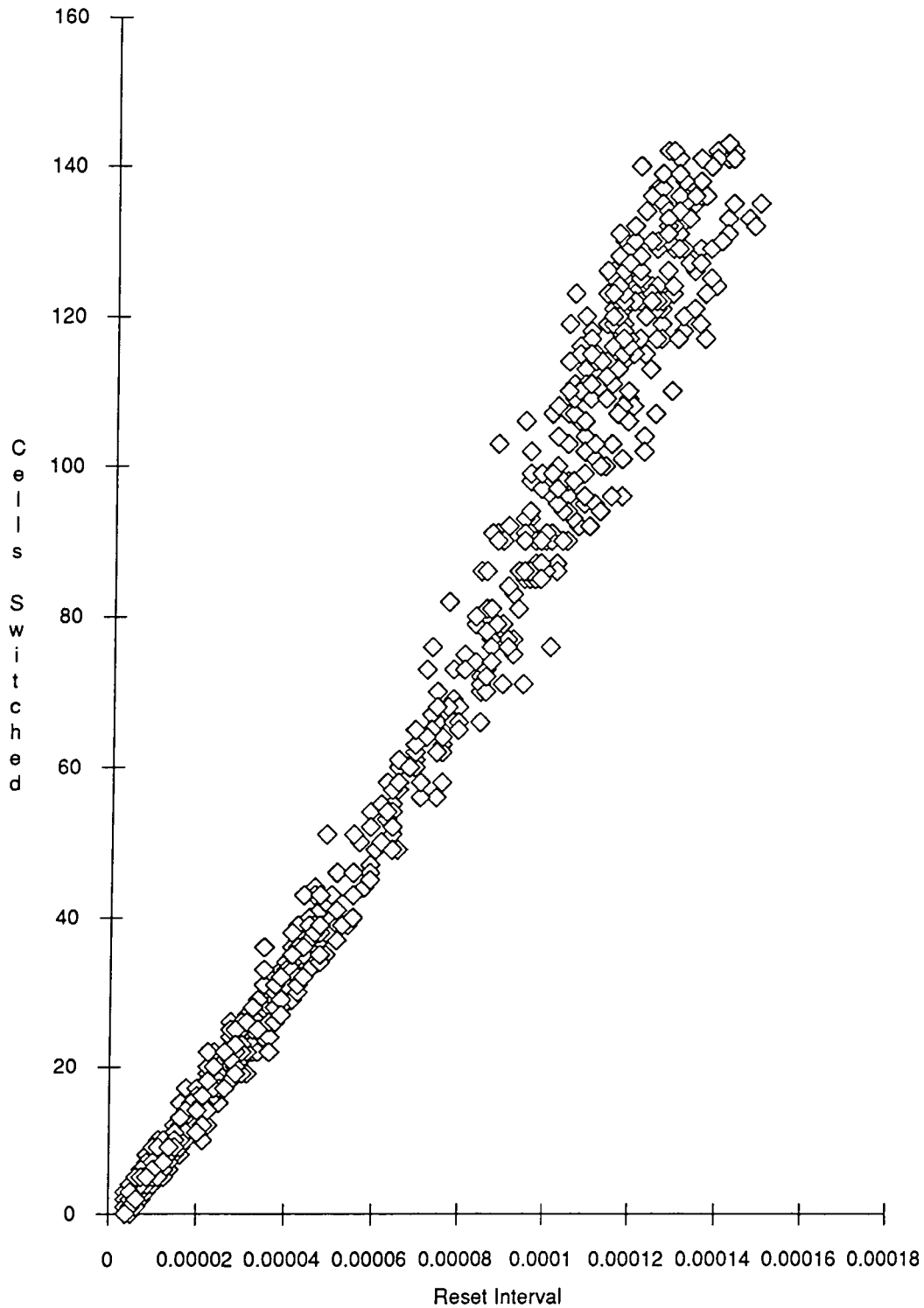

Figure 35: Graph showing the number of cells switched as a function of the size of reset interval for the Orwell protocol

to the simulator entailed modifying the code so that all changes to the status of a node were performed by a single sub-routine.

To take advantage of the knowledge of the state of activity of the ring thereby obtained, one small approximation is needed, namely that the delay before switching the following cell can be determined as the current one is being switched (in practice this will mean that the algorithm will slightly under-estimate the activity on the ring since nodes may well become active while a cell is being carried).

In addition to this a further simplification can now be made to the model without any loss of accuracy: if all the nodes are either idle or paused, then there is no need to do any inspections on the ring and, hence, time can be saved.

```
reset ring
while ring is not idle
        if there are active nodes
                generate a random node number
                while (node is paused or idle) and there are unchecked nodes
                        check the next node
                endwhile
                switch the cell
                next status value
                calculate delay based on number of active nodes
        else
                calculate delay when ring is idle
                next status value
        fi
        wait for delay
endwhile
```

An exact value for the average hold time as a function of the number of active nodes is very difficult to calculate. The following values were used to 'test out' the model, and seem to give reasonable results. When the ring is totally idle,

the delay is simply the time it takes for a slot to go around the ring, divided by the number of slots, t/KR. When a slot is seized it is held on average for half a rotation, so for KR slots the average hold time is $t/2KR$ (cf. equation 19, the fact that the slot cannot be seized until the following node is now accounted for by the search period). The search period is proportional to the number of *idle* nodes, and any particular slot will, on average, be half way to that node, giving the proportion of a rotation spent searching as $1 - a/N$ if there are a active nodes on the ring. The total delay is simply the sum of these two parts,

$$\begin{aligned} \tau &= \frac{t}{2KR} + \frac{t}{2KR}\left(1 - \frac{a}{N}\right) \\ &= \frac{t}{KR}\left(1 - \frac{a}{2N}\right) \end{aligned} \tag{23}$$

5.4.2 Results

The model was simulated over a period of 1.0 seconds after a warm-up time of 0.5 seconds, using 'totally idle' resets. Figure 36 shows that the reset interval is now much more closely matched to the original protocol and, in particular, the maximum load is almost identical. Figure 37 shows that, while queueing is now much more closely matched, there is still a factor of two difference on the results obtained so far.

Finally, figure 38 shows the total processing requirements for simulating the Orwell protocol and the third model (warm-up time and running time); it can be seen that, for any particular carried load, the model is marginally faster (though for very high offered loads the fact that the model still accepts slightly fewer calls means that the total simulation time is likely to be longer). To assess in detail the contribution to simulation time added by the nature of the Orwell protocol more simulations are required at very low loads. It is notable that the simulation time requirement is not linear with respect to the carried load, but increases significantly as the load carried increases; presumably this can be attributed to the processing of failed call attempts.

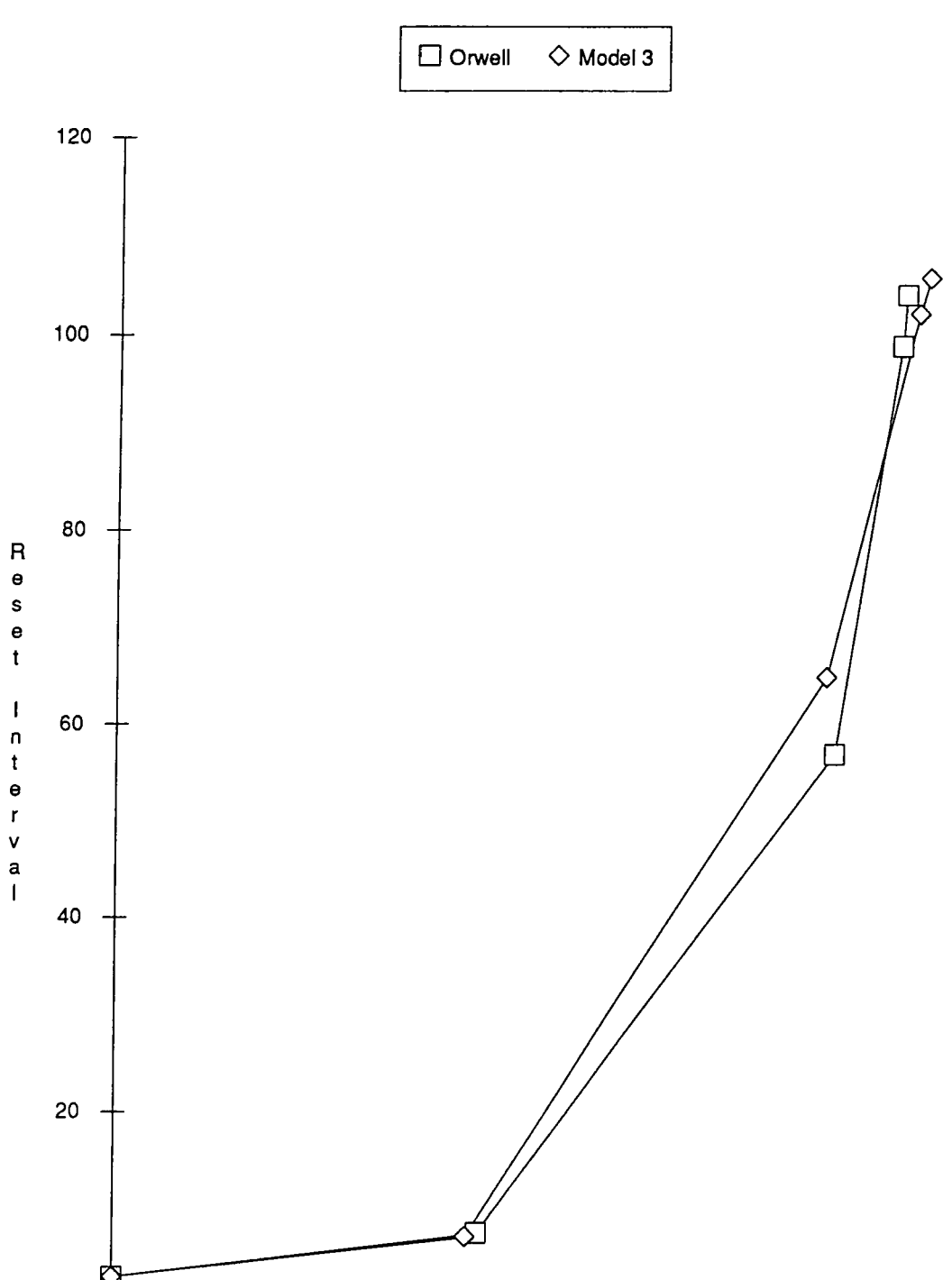

Figure 36: Graph showing the mean reset interval (in μs) against carried load for Orwell and the third model using 'totally idle' resets

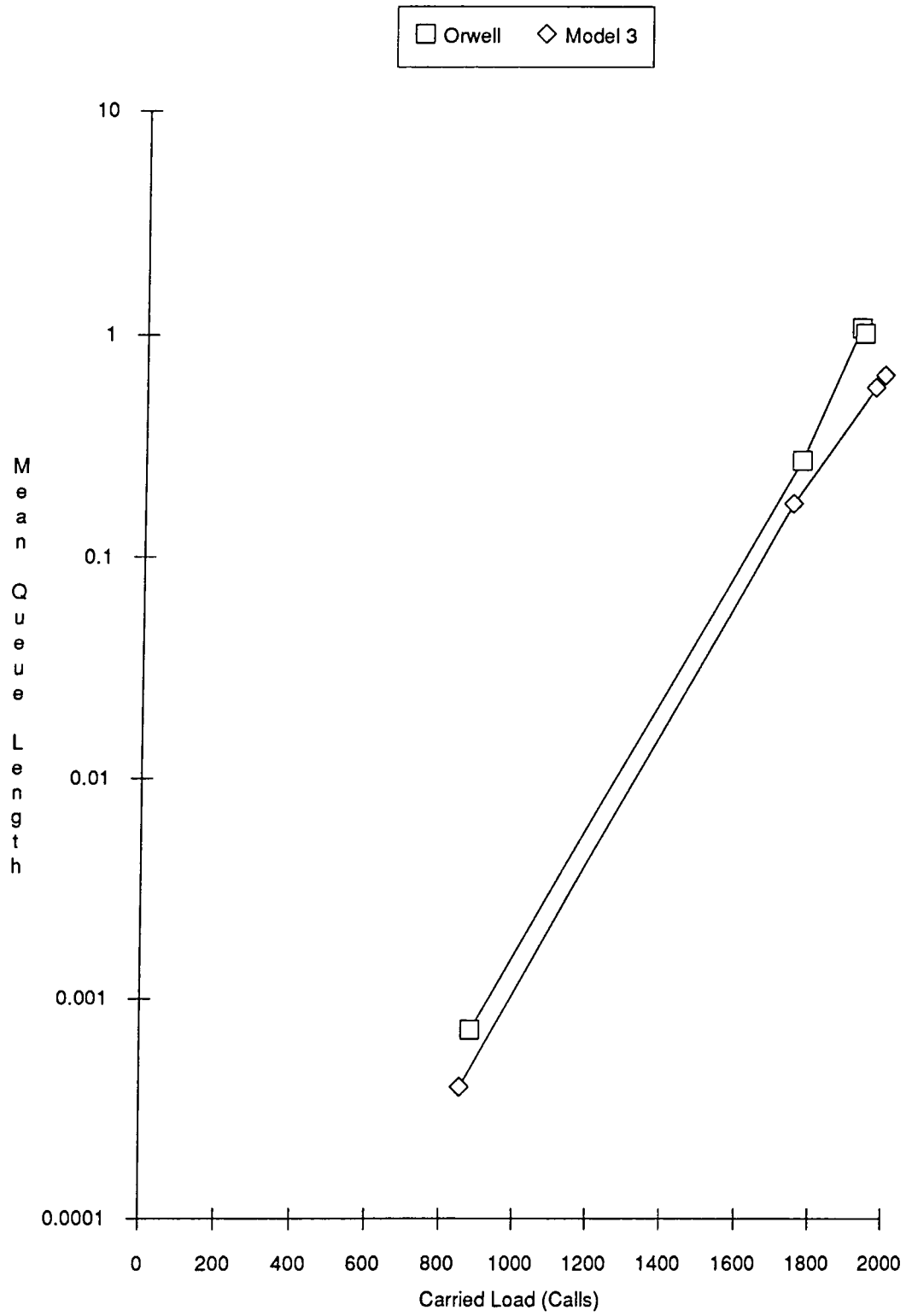

Figure 37: Graph showing the mean queue length against carried load for Orwell and the third model using 'totally idle' resets

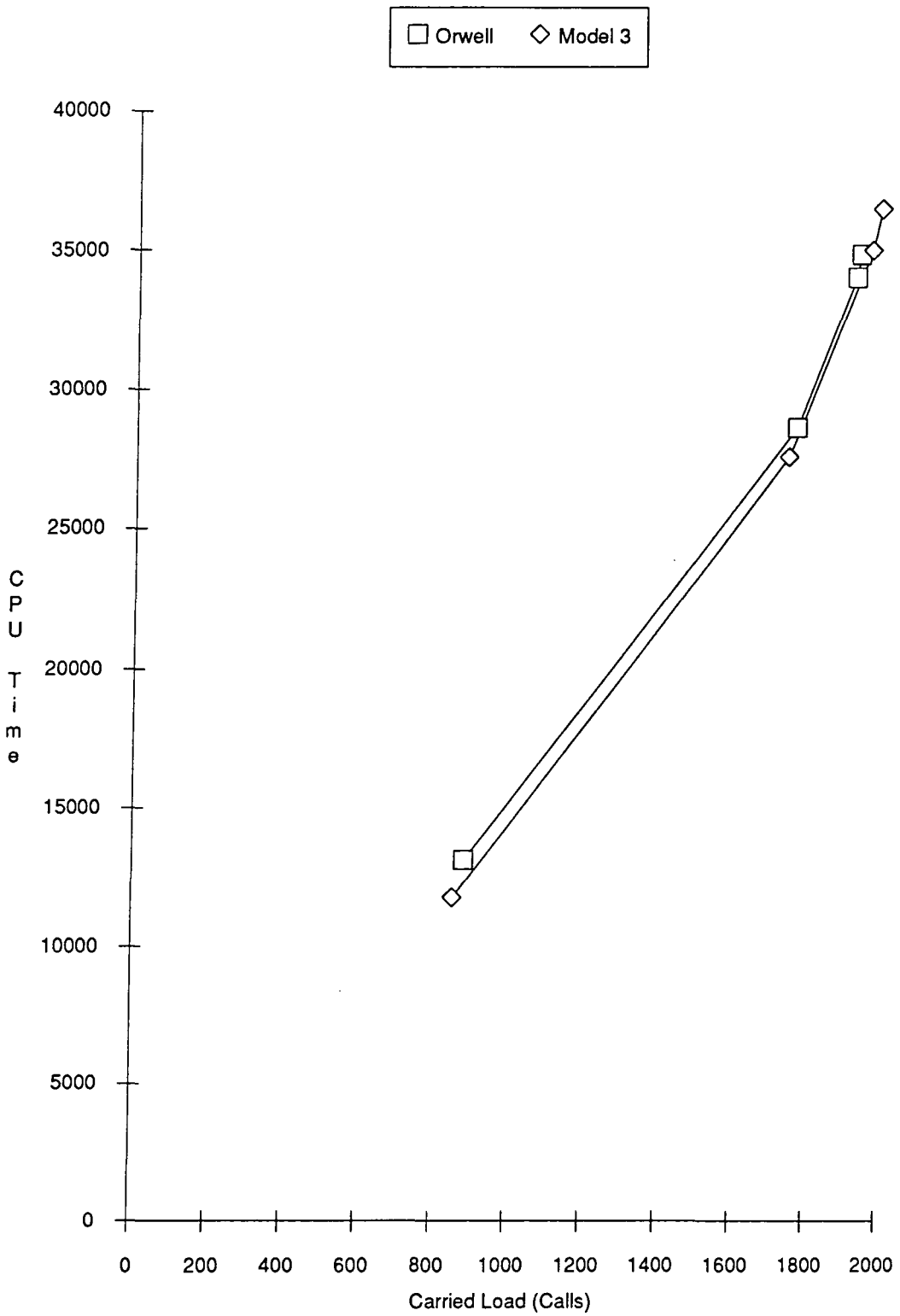

Figure 38: Graph showing the processor requirements on a Sun 3/50 work-station, in CPU seconds, as a function of carried load for the Orwell model and the third model

5.5 Summary

The pressing need to try and reduce the amount of simulation time required for simulating large networks has led to several models being created that simplified the details of the Orwell protocol, whilst still trying to maintain its outward functionality. It was found that the behaviour of the ring had two contributory factors: a service time, during which the slots were carrying cells around the ring; and a search time, while they were looking for new cells to carry. The service time had a constant average, while the search time was a function of the instantaneous load carried by the ring.

A detailed insight has been obtained into the behaviour of the Orwell protocol and, in addition, some reductions in the simulation time required have been achieved. However, it was decided that the model could not be incorporated into the A.T.M. network simulator without much further study.

Chapter 6

Support of Mobility using A.T.M. Techniques

MOBILE TELEPHONY has undergone phenomenal growth since the first truly flexible systems were introduced in the first half of the 1980's. These, for the first time, allowed both incoming and outgoing calls, and enabled the subscriber with a hand-held, portable, telephone to wander almost at will anywhere in the coverage area whilst the call was in progress. Support for a practical density of traffic was made possible by dividing the coverage area into small zones, called cells[1] (figure 39), and by re-using the scarce frequency spectrum at regular intervals; unrestricted wandering thus required that the transmission and reception frequencies had to be changed as a mobile moved from one radio cell to another. The rate of growth of these services, however, has meant that despite using smaller and smaller radio cells the network has already reached saturation in some areas; radio cells cannot be made still smaller using the technology of current networks due to the interference between them and the minimum transmission power. Introduction of digital transmission methods should reduce co-channel interference and allow better re-use of frequencies.

The use of smaller radio cells, however, poses a further problem: hand-offs from one frequency to another have to occur more often. When this is coupled with the fact that signal degradation, for digital transmission at the reception boundaries of a radio cell, is extremely rapid and that smaller cells are necessitated

[1]Not to be confused with A.T.M. cells: to avoid confusion the zones in mobile communications will be referred to as 'radio cells'.

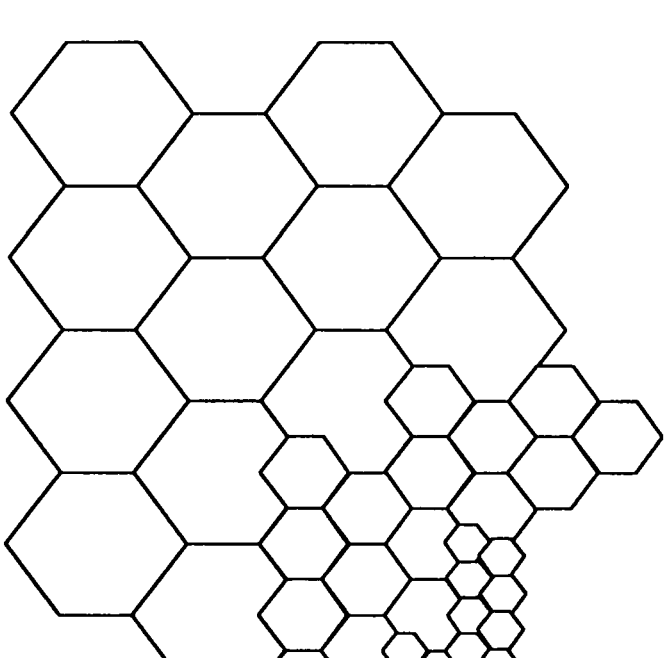

Figure 39: Radio cells covering a region. In areas where traffic is expected to be higher the cells are made smaller so that the re-use interval can be increased.

by higher load then the hand-off becomes both more common and time critical. For example, an express train, travelling at 200 kilometres per hour through the centre of cells 100 metres in diameter, would require each mobile on the train to perform a hand-off approximately once every two seconds. With current implementations, the hand-off is one of the most likely reasons for a call to be dropped by the network; although lack of a new channel is one of the most common causes for this, signalling failures are also a problem.

New forms of mobility are also being proposed. One example is for a customer to require access to a range of computer terminals each with different features: the customer would require that he be able to move from one terminal to another and for a connexion to a remote machine (or machines) to be already established with all the appropriate information that was available at the previous terminal; in this case the active point of the call would move from one terminal to another, but information about the status of the call must be transmitted to each of the

possible end-points at all times.

This chapter investigates a proposal that would implement mobility on an A.T.M. network, providing both better hand-off reliability and the new forms of mobility that are being proposed by using a footprint technique to perform a multicast delivery to multiple end-points in the network. Features of the local network are covered first, followed by the basic protocol that is required to enable the footprints to track the active point across the network. The protocol was implemented on the A.T.M. Network Simulator and the modifications required to permit this are then described. The final section uses results from running the simulator to indicate potential weaknesses in the protocol and to suggest ways in which these might be overcome.

6.1 The Local Network

The change from basic telephony to an I.S.D.N. was possible without the need to replace the existing local network: the copper pair was found to be able to carry more than twice the amount of information for which it had been originally designed. In the move to a B.-I.S.D.N. a new local network is unavoidable: the higher access rates demanded of such a network simply cannot be carried on a copper pair. Fortunately, in some areas the copper pairs are now reaching the end of their serviceable life and replacement is now becoming necessary. To provide a dedicated glass fibre to every house in the country would be overkill, since the full bandwidth of the fibre could never be fully exploited. Instead, a single fibre connexion to the exchange can be shared with several households by use of either a loop that visits each building, or by use of a tree structure and passive optical splitters: the latter technique is known as a Broadband over a Passive Optical Network (B.P.O.N.) [67, 68] and is one of the favoured approaches (figure 40).

One of the aims of the B.P.O.N. is to reduce the amount of electronics required in the street 'cabinet': the passive optical splitters copy the signal from the exchange to each customer's fibre, and interleave the signals from each customer

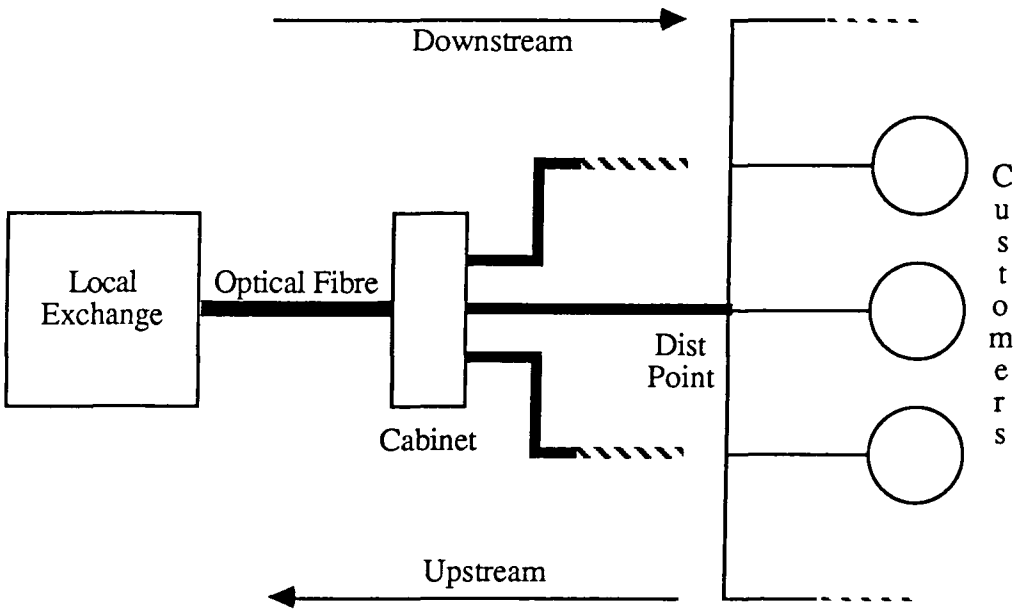

Figure 40: Tree structure of a B.P.O.N.. Both the cabinet and the distribution point contain passive optical splitters so that no additional electronics and lasers are needed.

for transmission to the local exchange; the limit of thirty-two customers is based on a power-budget for the transmitting laser. The broadcast feature of the system makes it well suited to distribution services such as television. Further, a load-control mechanism, like the one used in Orwell, can be used to allow dynamic allocation of both the up-link and down-link bandwidth for the A.T.M. connexion: a 150 Mbit/s channel split between 32 customers gives an average allocation of approximately 4 Mbit/s, but it is expected that there will be a high probability of obtaining much greater capacity on demand (for example, 15 Mbit/s) and that peaks approaching the full bandwidth of the channel should be possible.

The broadcast feature of B.P.O.N., as will be seen from the next section, makes it well suited for use as a final distribution mechanism to the base stations of a mobile network. An alternative approach could use ring technology, Orwell for example, as a metropolitan area network using a broadcast technique to achieve the same results: the Orwell protocol provides for broadcast addressing as part of its standard header; and multicast addressing (sometimes known as group addressing) as part of the media-access control layer.

6.2 The Basic Protocol

In conventional mobile systems[2] a call to a mobile handset starts with the network interrogating the home location register (H.L.R.) of the destination mobile. This register, amongst other things, contains the last known location of the called party and this is used as a starting point when paging for the mobile (this information is kept up-to-date by occasional registration by the mobile). If the initial page fails then successively larger areas can be paged until either the mobile is found or it is deemed to be unreachable. Once the mobile has been precisely located a connexion can be established in a conventional manner, very similar to that used in non-mobile telephony (for example, POTS). Establishing a call from a mobile to elsewhere is much more straightforward, since the paging phase is not required.

As a mobile approaches the edge of the coverage area of a radio cell, the base station, which is responsible for measuring signal strength, can either instruct the mobile to transmit at increased power or, if the power limit of either the radio cell or the mobile has been reached,[3] it can decide that a hand-off to a neighbouring cell is required. The base station instructs its neighbours to monitor the signal strength from the mobile and the one with the best reception is elected to handle the call; a connexion is established through the network to the new base station, either by re-routeing the old one or by establishing a new one, and the mobile is then instructed to start transmitting and receiving on new frequencies. Terminating the call is very similar to conventional telephony. In order to simplify network management and billing procedures a single node is selected when the call is initiated to form an 'anchor node'; this node remains the same and the circuit passes through it at all times regardless of where the calling or called parties might roam during the call. Alternative approaches to letting the base

[2]A general implementation is intended here, but the approach is based loosely on that used in TACS (Total Access Communication System), the analogue system currently in use in the U.K.

[3]Radio cells have limits to prevent interference, the mobile for safety and power consumption considerations

station determine that a hand-off is required include the mobile sending a request to the base station that a new one be found, but at present almost all systems require that the base station itself select where, and eventually when, the hand-off should occur; in this way tighter control can be maintained over allocation of radio channels and other network resources.

6.2.1 The Time-critical Hand-off

The hand-off is clearly the most important distinguishing feature of the mobile system, so it is important that it works reliably and quickly: with the existing TACS, for example, there is a noticeable break in communication each time a hand-off occurs; for a digital system that may well be used for carrying loss-sensitive data this is clearly unacceptable and steps must be taken to ensure this does not happen. Further, from detecting that a hand-off is required to completing the process is a time-critical procedure; failure to complete it sufficiently quickly can result in the mobile moving out of communication range and the call being lost. The problem is exacerbated by Rayleigh fading, which is a complicated function of the distance between the mobile and the base station and is caused by multipath interference of signals reflected from buildings; this is in addition to normal fading due to distance and the basic terrain between the transmitter and the mobile [69].

The following steps have to be taken once it has been decided that a hand-off is required:

1. Poll surrounding base stations to establish which is receiving the signal most clearly;

2. Create a new connexion across the network to the new base station;

3. Instruct mobile to start transmitting over new radio channel;

4. Clear old connexion.

The time-critical parts are steps one and two: the first step involves substantial amounts of signalling across the network to establish which of the surrounding base-stations can best receive the mobile; the second requires further signalling (this time end-to-end) to establish the new channel before the hand-off can take place. If either of these can be simplified, or moved outside of the time-critical period then the speed of a hand-off can be significantly improved.

Whilst not strictly essential to the implementation of the new mobility protocol, moving the decision about initiating a hand-off from the base station to the mobile could significantly reduce the amount of signalling required: the mobile is able to monitor each of the neighbouring base stations using the signal strength of the common signalling channels (each base station transmits continually using such a channel to enable mobiles calling up the base station to detect collisions on the reverse link); the selected base station can then be signalled by the mobile indicating that it wishes to transfer control of the call. A further advantage of such an approach is that it may be possible to continue a call even when contact with the previous base station has been completely lost: the main disadvantage is that the intelligence has to be passed down to the mobile, making optimizations from global knowledge hard to implement, and performance tweaks to the hand-off algorithm impossible once the network has been installed.

6.2.2 Footprints

The time delay associated with establishing a connexion to the new base station can be minimized by creating a passive connexion in the time before a hand-off is required. Since, at this stage, the appropriate base station is not known, passive connexions have to be established to each of the potential hand-off points, forming a 'footprint' that surrounds the current position of the mobile[4] (figure 41).

[4]The basic footprint technique was originally proposed by John Adams of B.T. Research Laboratories; to the author's knowledge, there is no formal definition of the procedure currently available in the literature.

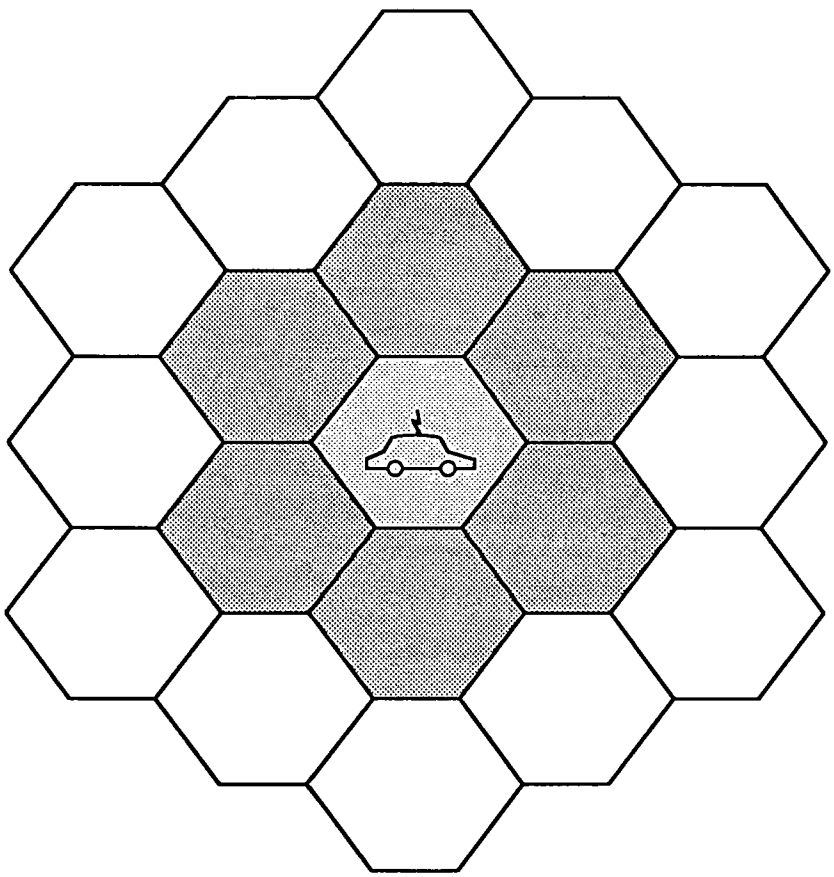

Figure 41: Footprint about a mobile station. The dark-shaded areas are the periphery of the footprint, but the mobile is normally in communication with the radio cell at the centre. When the mobile moves to one of the cells in the periphery the call information is already established.

When a hand-off is required, the connexion is already established and simply has to be activated, a much faster process than establishing one that is new: once control has been transferred, the old connexion and the unused passive connexions can be cleared and a new set of passive connexions established around the new position of the mobile; the footprint tracks the mobile in the manner of a spotlight, except that the movements are discrete jumps instead of being continuous. A passive connexion receives all signalling and data from the remote end of the network, but transmits none in reply: only the active connexion transmits; performing a hand-off requires simply that the active connexion be changed.

6.2.3 Implementation of Footprints using A.T.M.

Footprints can be easily implemented on an A.T.M. network using a multicast facility. Connexion management across the network can be simplified by making the network level connexionless (see section 4.2.2); cells travelling across the network are destination addressed and this can be used to generate the appropriate multicast duplications that are required to cover the footprint. Each footprint is labelled by the destination address of the radio cell at its centre. Normally the entire footprint will lie entirely in the region of a single local distribution system (for example, a B.P.O.N. or an Orwell ring used as a MAN) so no multicasting is required within the main network (figure 42): occasionally a footprint might be split between two or three distribution systems (it should never be necessary to have more than three, as in figure 43); if all of the local networks are connected to the same local exchange then a multicast is needed at the local exchange; otherwise a multicast is needed one step further back in the network (it should never be necessary to go further back than this) making it possible to implement broadcasting on a network of trunk and local exchanges with the local exchanges dual-parented.

By structuring the multicasts in a hierarchical manner it is possible to create a single tree to cover each footprint. Provided that all cells destined for a footprint are routed via the same tree root then it is possible to guarantee that loops leading to infinite duplications are avoided. Simple duplication rules can be applied to ensure that multicasting works correctly even when the data cells originate at a leaf within the footprint: a cell should be duplicated back from an exchange to a leaf (local network) within the footprint, but should not be duplicated back to an exchange from which it has already come. Figure 44 shows how this rule works in practice, the local-broadcast networks within the footprint are shaded grey (local networks w, x and y) and since these lie on two, separate, local exchanges a multicast is required at the trunk-exchange level. The general case is shown in figure 44a; cells normally arrive at node A from exchanges elsewhere in the network and are multicast to nodes B and C; cells arriving at B are transmitted

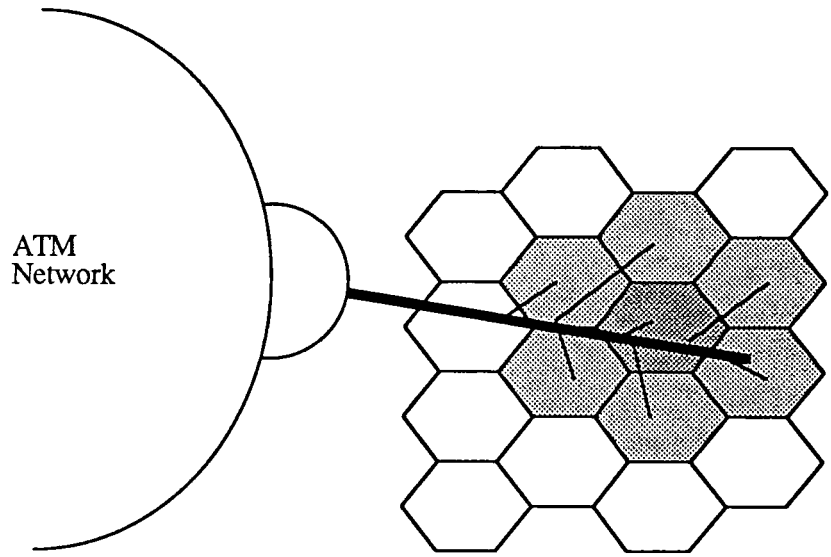

Figure 42: A footprint can normally be covered by one local-broadcast network from the ATM trunk network.

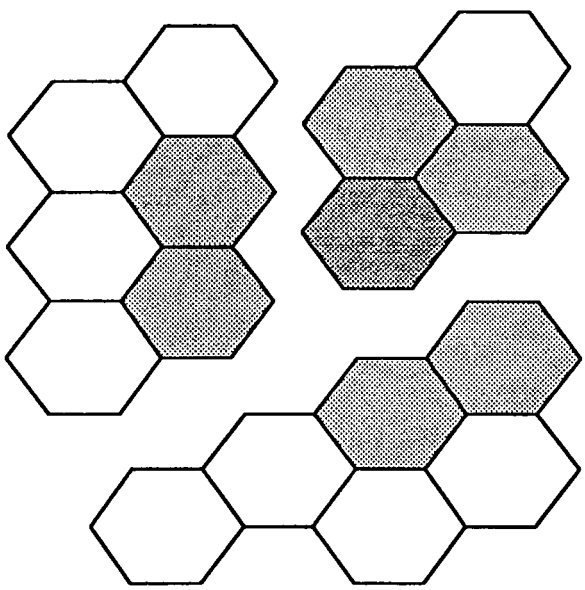

Figure 43: In the worst case it should not be necessary for a footprint to be divided between more than three local-broadcast networks.

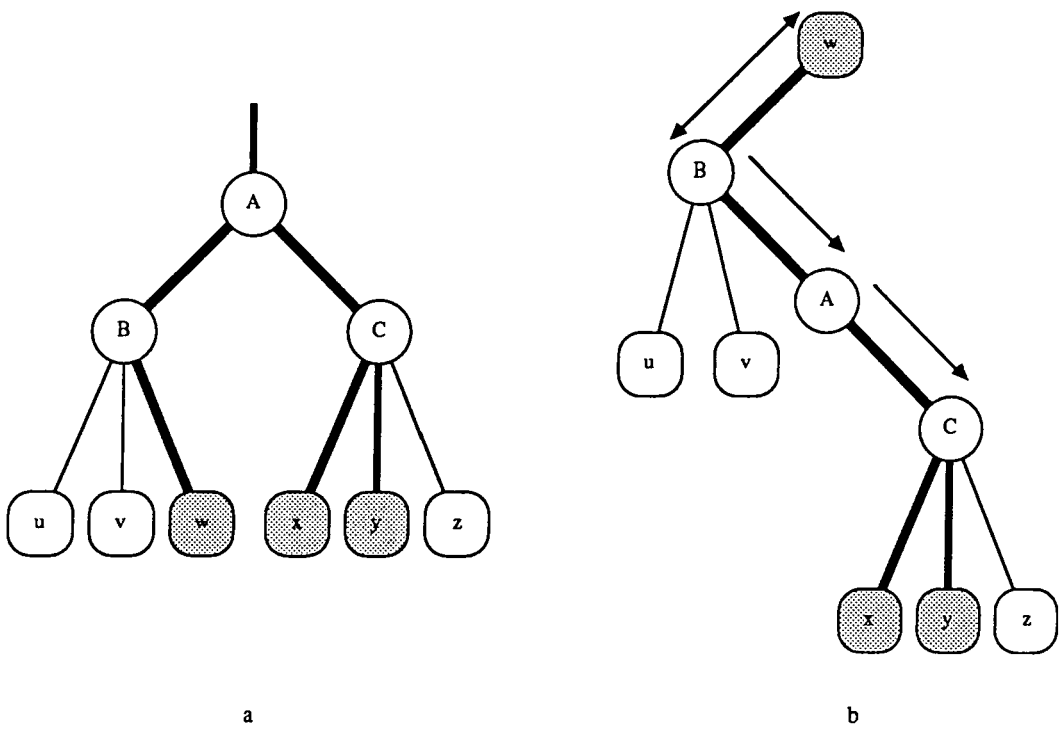

Figure 44: A single tree-structure is defined for the A.T.M. Multicast. a) The normal broadcast route, for cells originating outside a node connected to a node in the footprint. b) A special case when the transmitting mobile is in the periphery of the footprint.

solely to the local network w, while those arriving at C are again multicast to local networks x and y. A peculiar case is where the cells originate from a local network within the footprint as shown in figure 44b; in this case, cells arriving from the local network w at the node B have to be duplicated back to w (since it is a leaf in the footprint) and also copied to node A; at A the cells have to be sent to C, but not back to B, and it is possible to cater for this since the cells arrived from B in the first place. To simplify construction of the routeing and multicasting tables it is quite important that variants are not required for each input of an exchange: by applying the duplication rule given above it is possible to see that a single routeing and multicasting table can be used for each node. In particular, using the example given, it is possible for node B to differentiate, sufficiently, cells destined for the shaded footprint that originated at w from those that originated elsewhere in the network and to treat them appropriately; this is

despite the fact that both bear identical destination addresses in the cell-header. Further consideration also shows that cells originating from local exchange v would also be treated correctly by B and multicast to w and A since neither was the origin of the cells.

6.2.4 The Basic Hand-off Message

Since the connexion to the new end-point in the network has already been established before a hand-off is required, there is no cross-network signalling during the hand-off itself. Instead, after the hand-off has taken place new passive connexions have to be established in the footprint of the mobile's new position. Further, since the network level is connexionless a new destination address has to be used to ensure that cells are sent to the new footprint. Finally, any passive connexions that then lie outside of the current footprint have to be deleted. The A.T.M. adaptation layer of each cell contains a connexion-identifier that is sufficient to resolve uniquely each call: since cells are multicast to a footprint, which overlaps with other footprints, it is necessary to be able to resolve between identical connexion-identifiers in different footprints; full resolution of the connexion requires use of the connexion-identifier in conjunction with the address of the destination footprint. When the destination address is changed, therefore, a new connexion-identifier must also be generated to ensure that the connexion can still be fully resolved.

The change of connexion identifier and footprint address is effected by transmitting a Hand-off Update message across the network to the remote footprint[5] that contains both the old and new connexion-identifiers, and instructing that, in future, cells should be sent to the new destination. A response message is generated by the active receiving point of the original message (it was multicast to a footprint, like all other cells); it contains the same information fields (corrected

[5]The end at which the hand-off occurred will be referred to as the local footprint, the other as the remote footprint. When considering situations where both ends are updating connexion information from hand-offs simultaneously the local footprint will refer to the end which started the process first.

as required, see below) but is addressed to the new footprint; this is sufficient to create the new passive connexions at each of the remaining points in the footprint. Finally, a message is sent from the new local active point to the old local footprint informing it that the new connexions have been correctly established and that the old ones can be released. Plate I shows the signalling involved in a simple hand-off: when the mobile moves from the old to the new controlling point a new connexion-identifier is generated and the hand-off message, with the fields shown in figure 45 is transmitted across the network. Reception of this message

Old connexion-identifier
Old footprint address
New connexion-identifier
New footprint address
Remote connexion-identifier
Remote footprint address

Figure 45: Fields in the hand-off message.

by the remote footprint causes all the remote tables to be updated with regard to the connexion-identifiers. Additionally, the active remote point generates the acknowledgement, which in this case is identical to the original message, and sends it back to the new local footprint; receipt of the acknowledgement by the new active point is taken as an acknowledgement that the signal traversed the network successfully, receipt by other end-points in the footprint is taken as an instruction to establish a new passive connexion for the call; the final action of the new active end-point is to generate a message that is sent to the old footprint to clear the old passive connexion. Plate II shows how the cells being transmitted from the remote end change connexion-identifier after the hand-off message is received.

6.2.5 Concurrent Hand-offs

It will occasionally happen that both mobiles will attempt to change footprints at the same time: more accurately, that both ends will be trying to update their

Explanatory Notes to Plates

In the following plates a diagrammatic convention has been adopted to clarify the representation of the signalling information. Each diagram is divided vertically into two halves: the left-hand side represents the local mobile transmitter and two of the transmitters in its covering footprint; the right-hand side represents the remote mobile. The only transmitters of interest during a hand-off are the old active transmitter and the passive transmitter that will become active as a result of the hand-off occurring: signals destined to the other passive transmitters will be identical to whichever of the two represented here is not active at a particular time; the actions taken by the remaining passive transmitters form a sub-set of those given here.

Within each side of the diagram two signal origins are indicated: the left hand representing the station that was originally active; the right that which was originally passive. Colours are used to indicate the origins of a signal chain in terms of the connexion-identifier used: all signals generated as a direct result of an initial message are associated with the colour assigned to the initial connexion-identifier. Vertical lines of colour within a transmitter indicate the existence of a session associated with the identifier at that point.

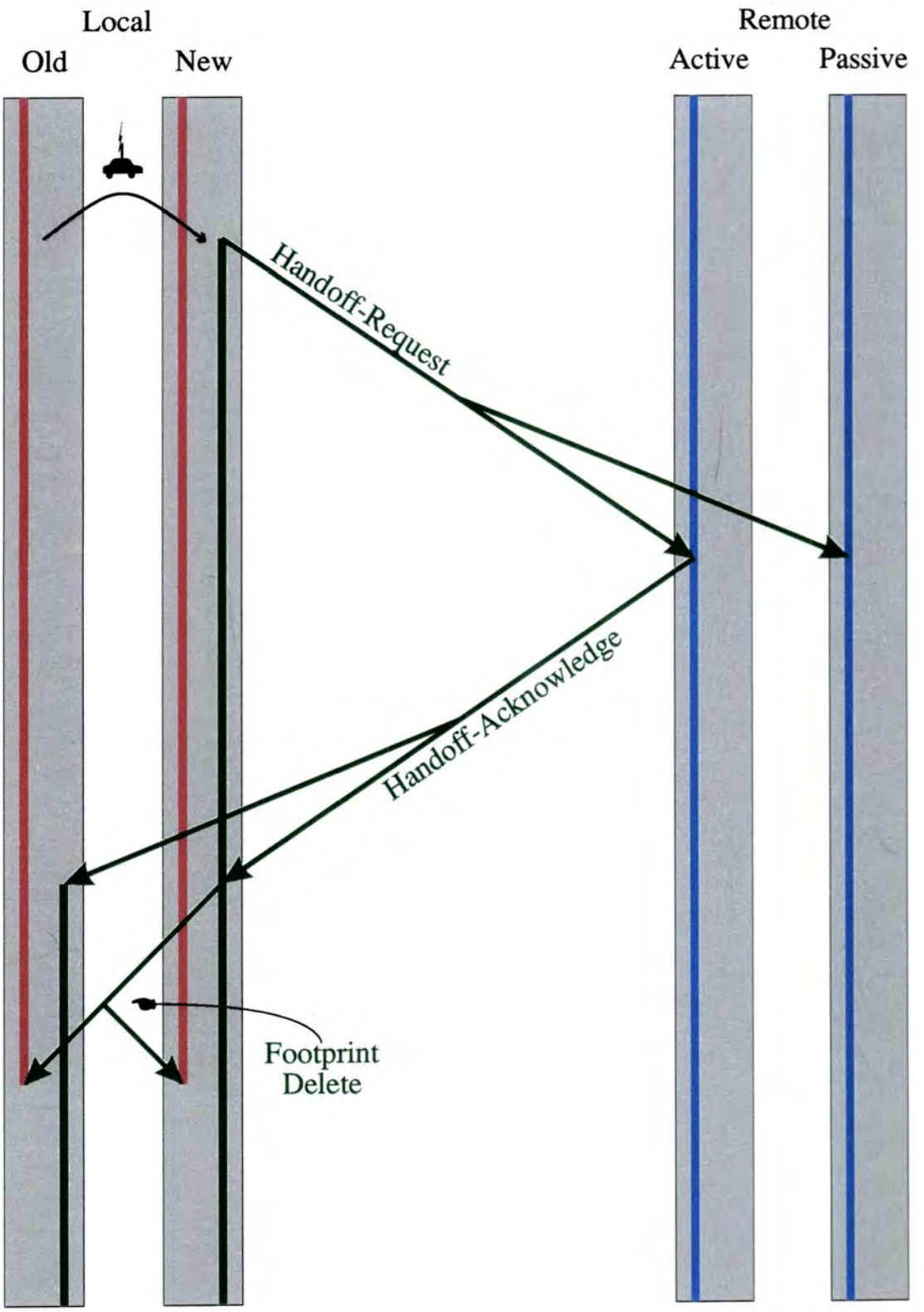

Plate I Basic hand-off message

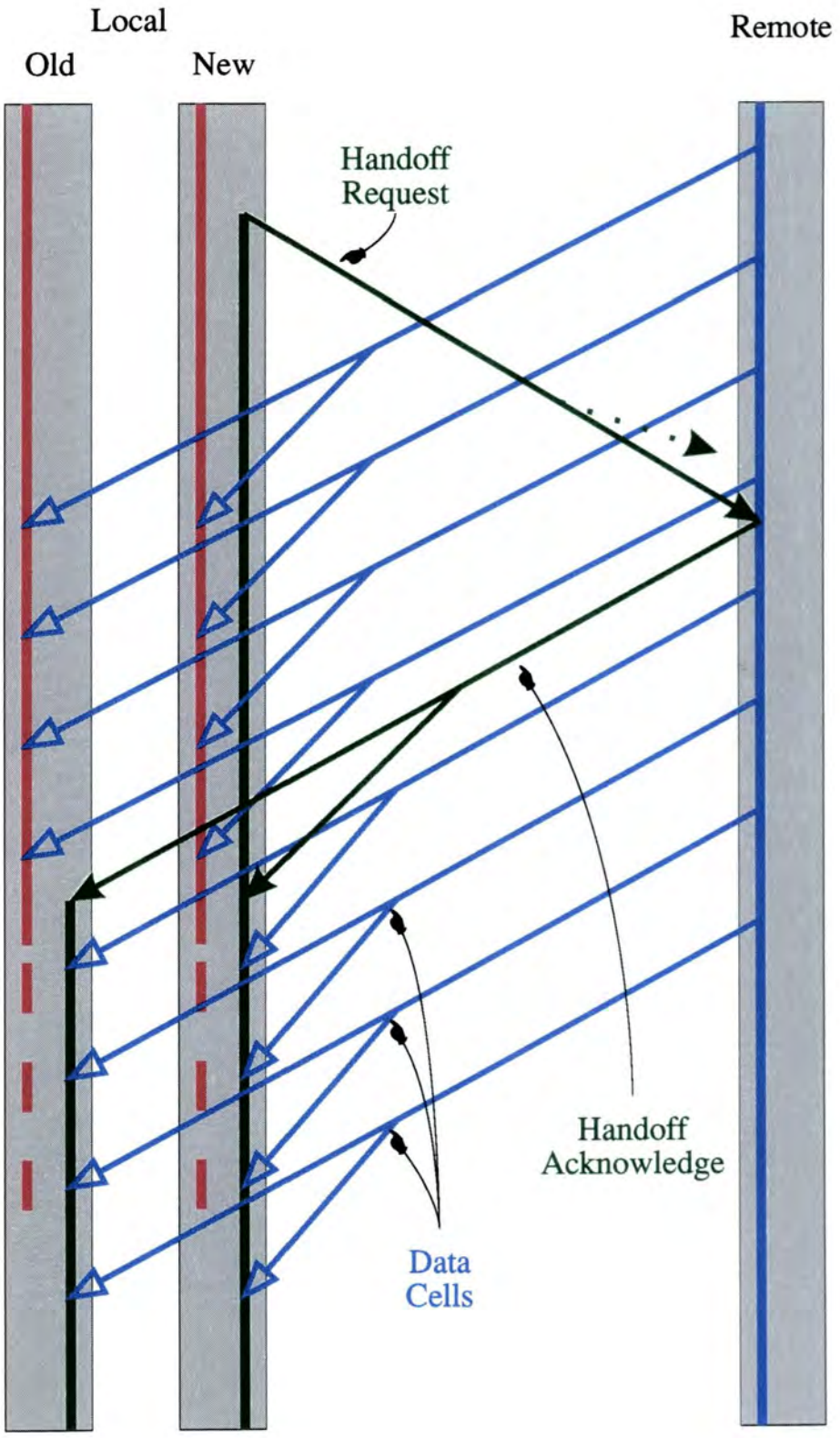

Plate II Data cells during hand-off

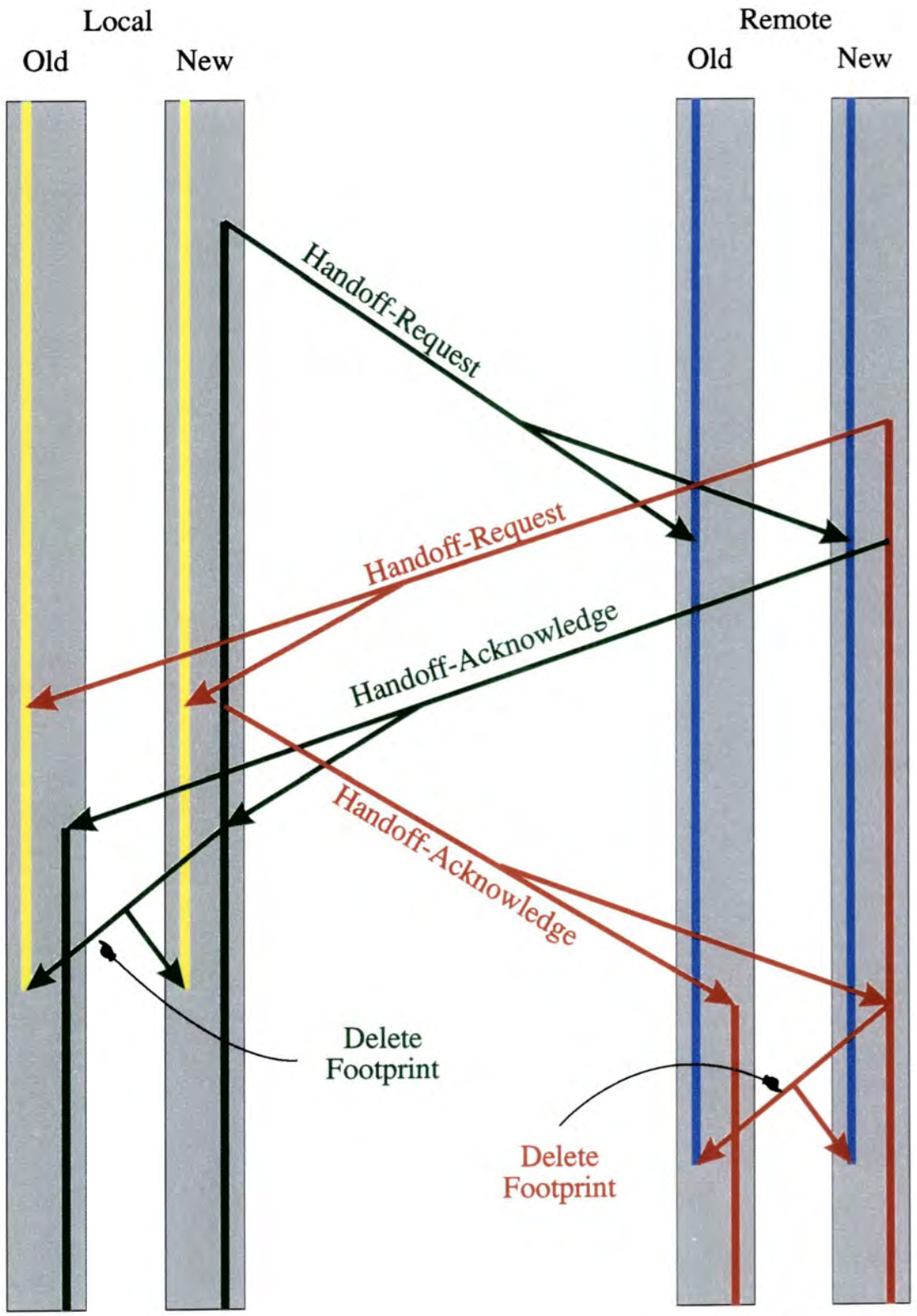

Plate III Concurrent hand-offs

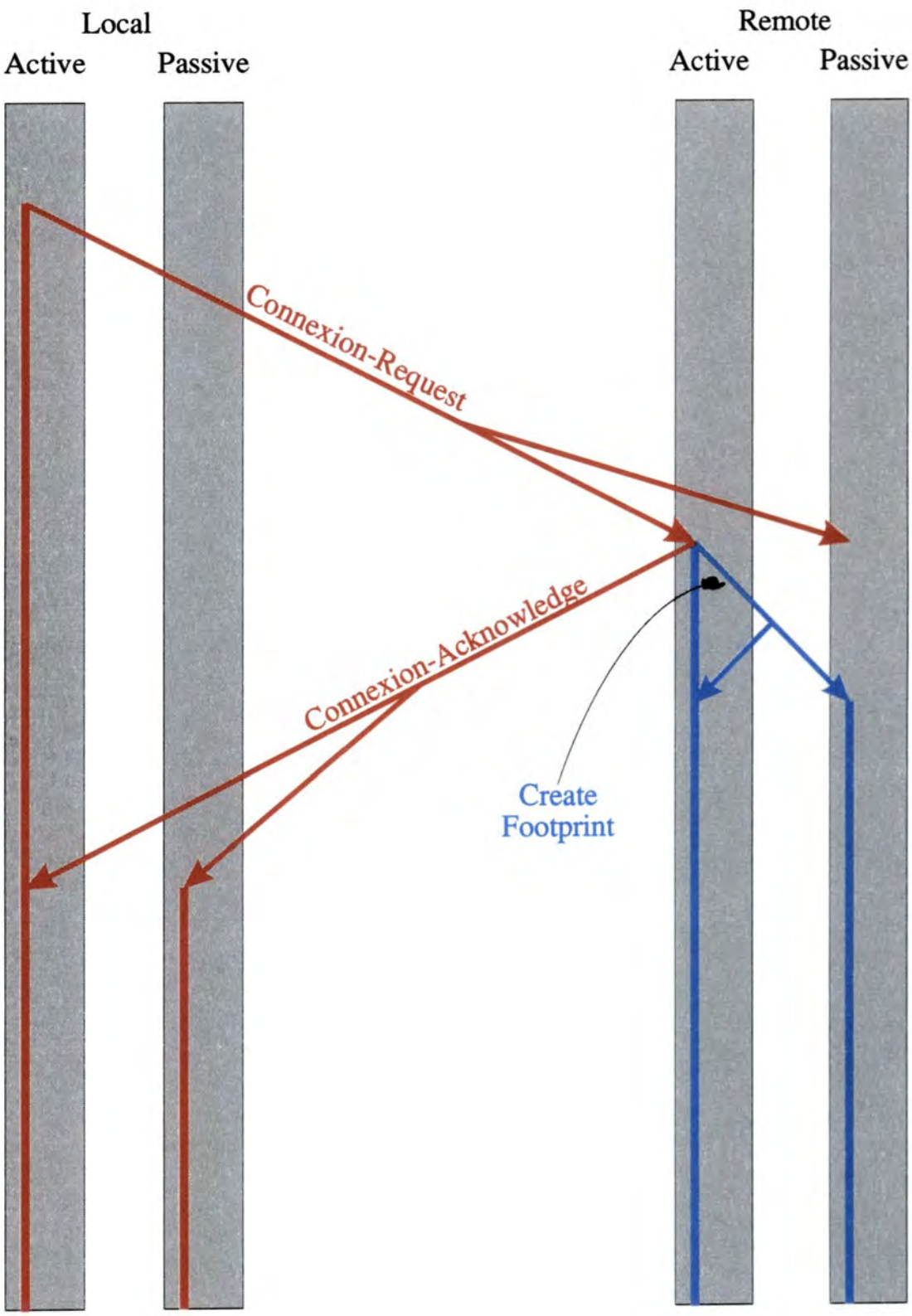

Plate IV Call set-up signalling

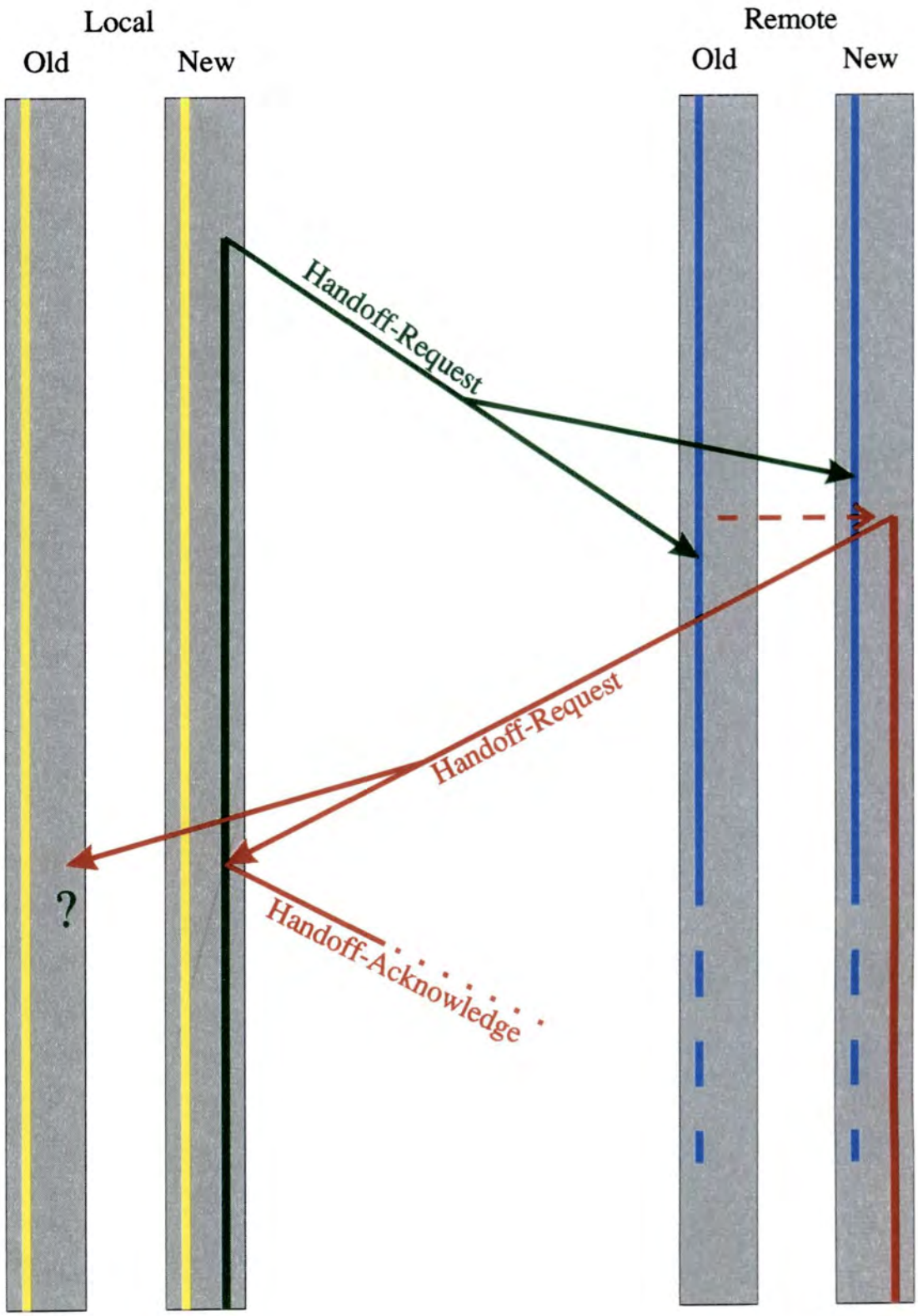

Plate V Loss of hand-off signal due to jitter

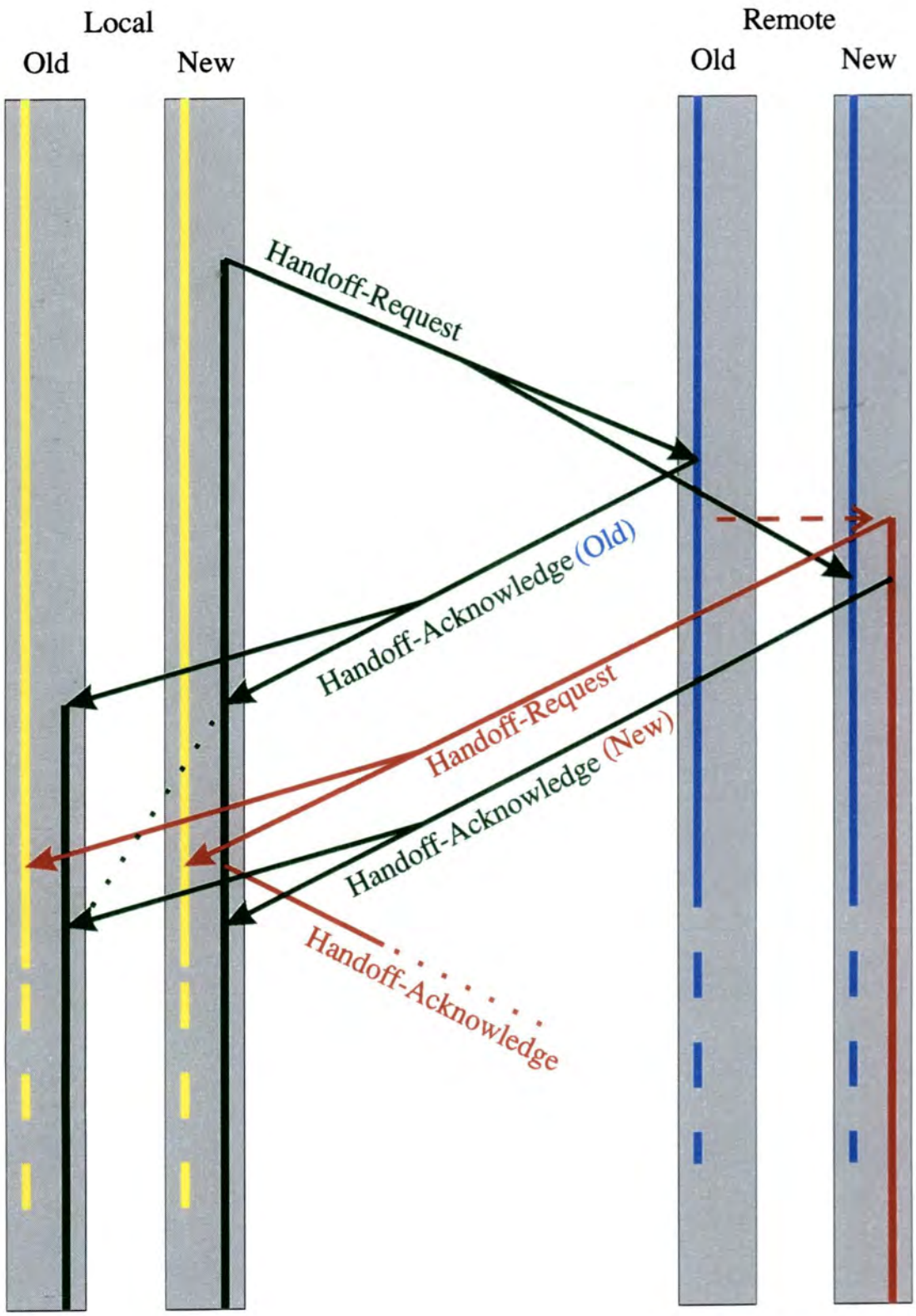

Plate VI Duplicate hand-off acknowledgements due to jitter

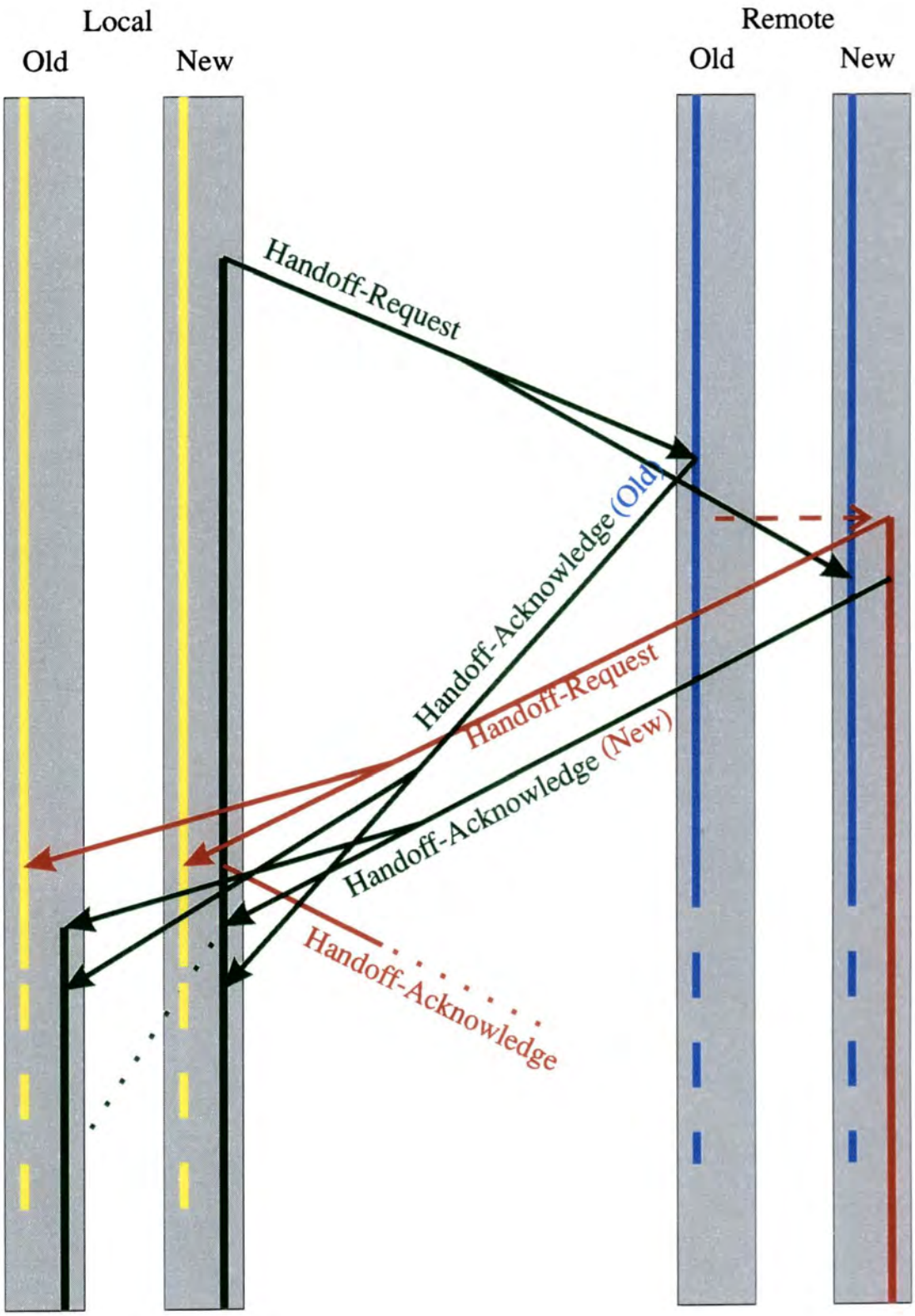

Plate VII Acknowledgement from old end-point over-writes correct status

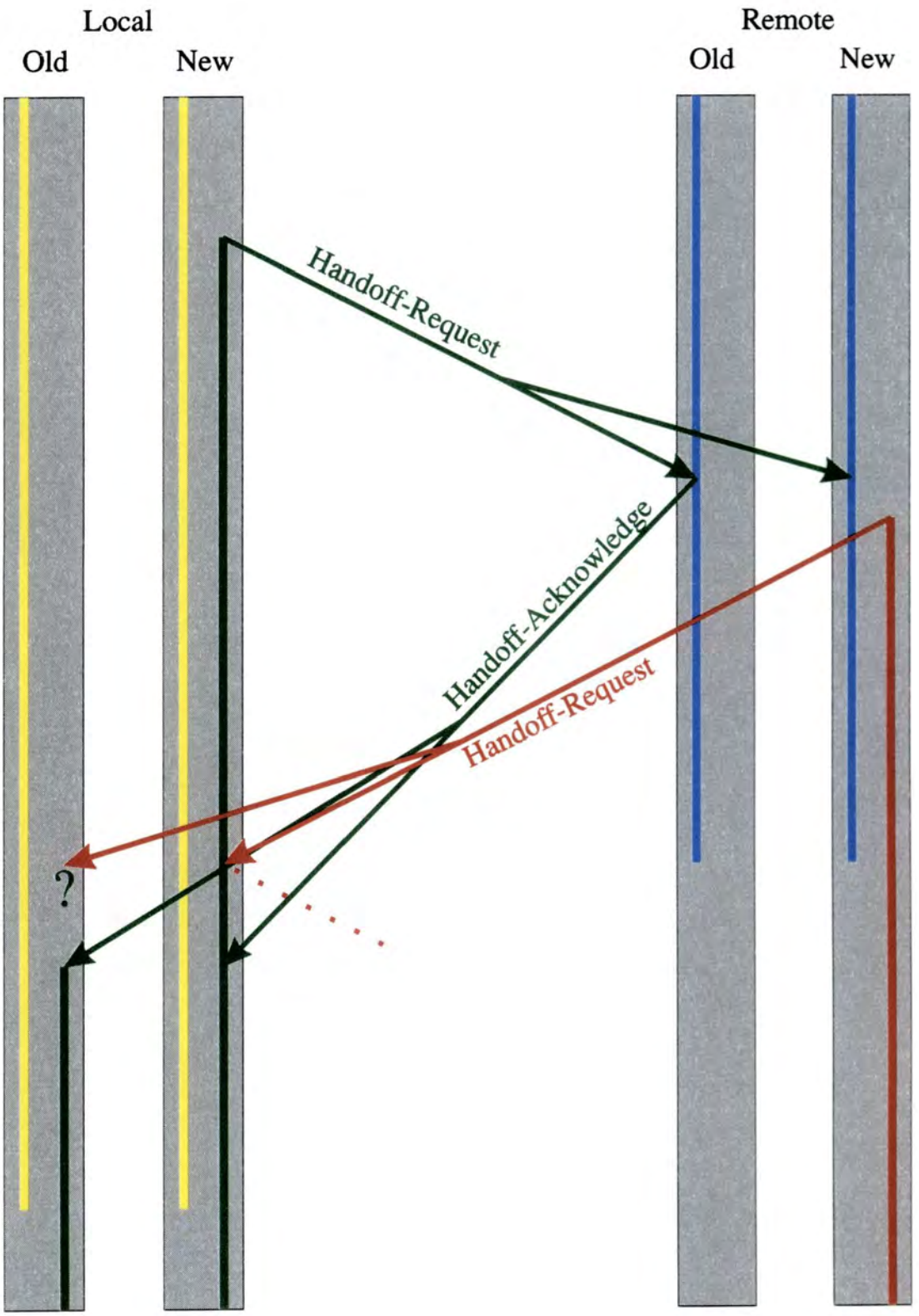

Plate VIII Hand-off request arrives before acknowledgement

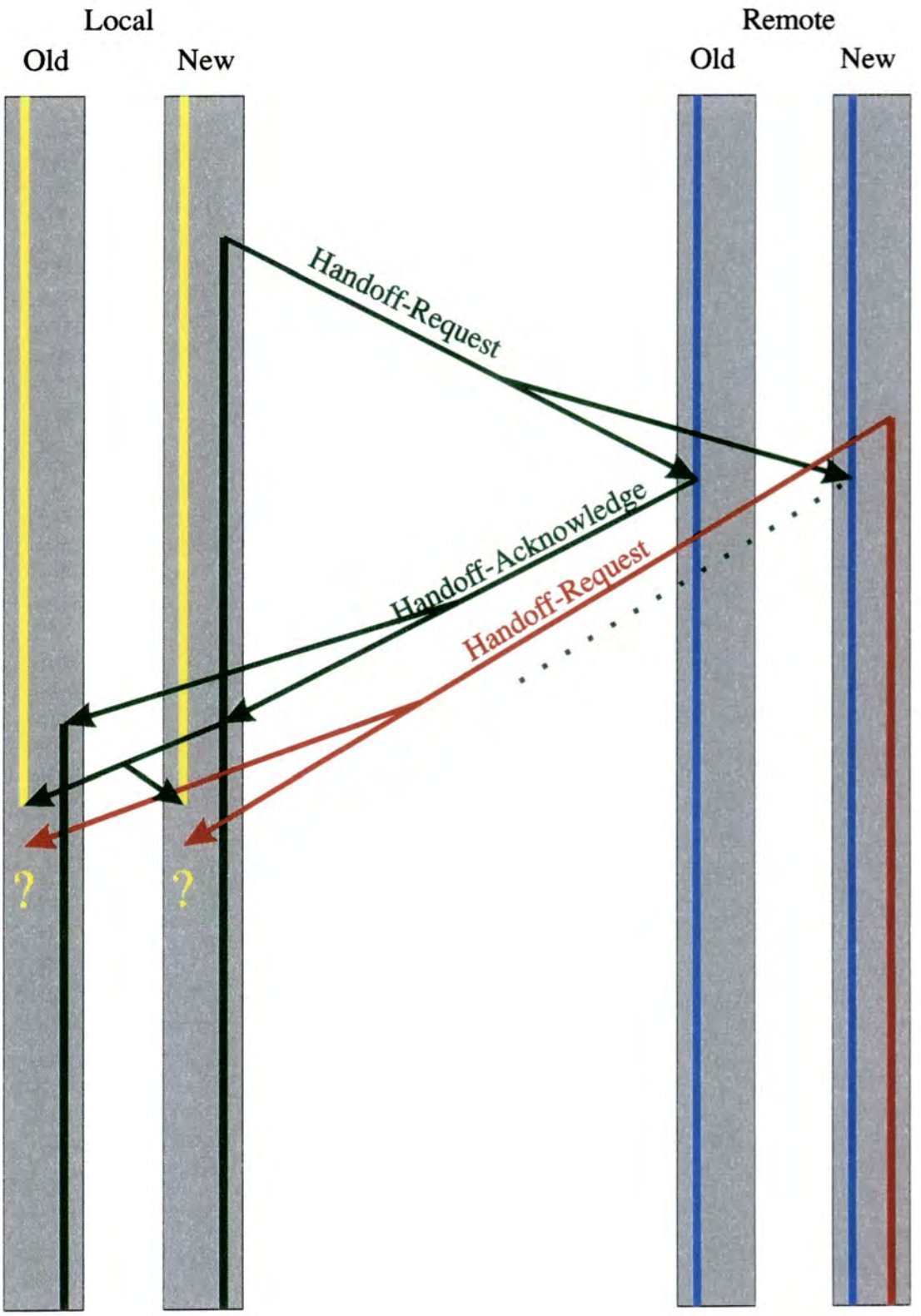

Plate IX Hand-off request to old footprint is delayed

footprints and connexion-identifiers simultaneously; when this occurs the initial hand-off messages are transmitted across the network towards the old remote footprint, and some care is required to ensure that the acknowledgement message comes from the correct place and contains the correct connexion-identifier information. Plate III shows the signalling messages for such a situation: the local mobile is the first to initiate a hand-off and generates a hand-off message to the remote mobile's footprint, but before it arrives the remote mobile also initiates a hand-off and generates a message to the *old* local footprint. On receipt of the hand-off message at the old remote footprint the currently active base station recognizes that the message arrived using a superseded connexion-identifier and generates an acknowledgement message that contains the corrected values; since it is the acknowledgement message that is used to establish the new local footprint the correct values will then be used; by the time the active local end-point receives the acknowledgement it will have already received the remote's hand-off message and updated its connexion-identifier table accordingly.

6.2.6 Data Continuity During Hand-off

When the mobile transfers from using one fixed point to another then the route that data flowing towards the mobile takes through the network changes and the time taken to negotiate the two paths is slightly different. Even when the path lengths of two different routes are the same the time taken to negotiate them will differ slightly because of queueing delays at the intermediate nodes. With any packetizing scheme for constant bit rate data some reassembly and retiming of the cells is required to eliminate cell jitter, which is caused by the queueing; and the same reassembly process can be used in the mobile to eliminate the differential path delay that is incurred from the change of end-point. There is, however, an additional hazard, resulting from the duplication process used within the network, which means that it is possible for a cell to be received twice, or possibly not at all. The path lengths and jitter delays combine to make slightly different end-to-end delays for two copies of the same cell, so it is possible for the

remote mobile to hand-off from one end-point to another at a time immediately after a cell has been received from the local mobile; it is then possible to start using the new end-point sufficiently quickly for the duplicate of that cell to be received from the new end-point: in this situation the mobile receives the same cell twice. Conversely, it is also possible to initiate the hand-off immediately before a cell arrives, and to start using the new end-point just after the copy of the same cell has been discarded because the end-point was in the passive state; in this case the cell is lost. The two situations are shown in figure 46.

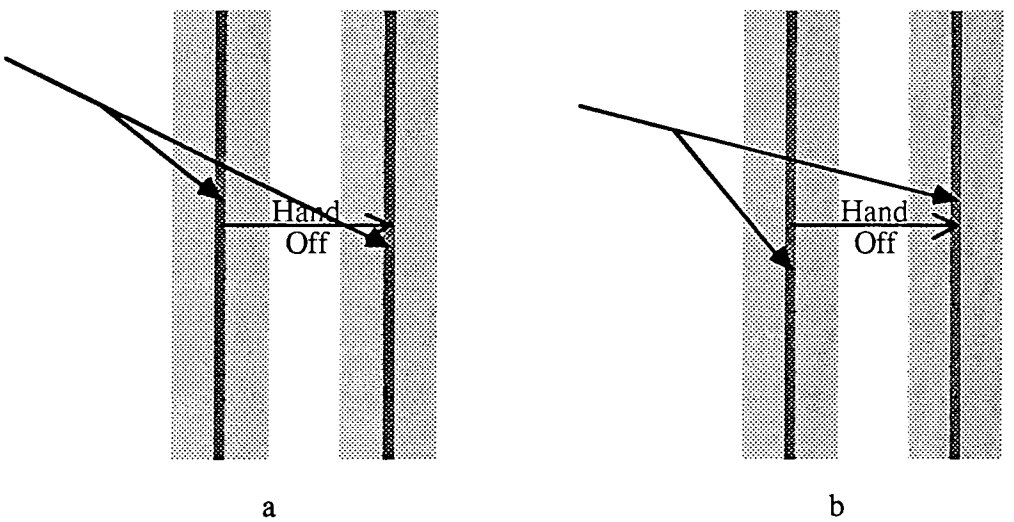

Figure 46: Cell duplication and loss during a handoff. a) Two copies of the same cell are received because the copy going towards the periphery of the footprint is delayed. b) The cell is lost entirely because the copy going towards the centre of the footprint is delayed until after the mobile has handed off.

The cell loss is clearly the more important problem to solve: receiving a cell twice can easily be detected, and duplicates eliminated, by using a two- or three-bit cyclic counter; once a cell has been lost it can only be recovered by retransmission, and this is not an option for delay-sensitive networks. Fortunately, the system can be easily modified so that this situation never causes cells to be lost at the expense of more duplications occurring: each passive end-point maintains a copy of the last data-cell that it has received, and immediately a hand-off takes

place forwards this cell to the mobile which can then eliminate duplicates as before. With this approach it is possible for the stored-and-forwarded cell to have a sequence number one less than that of the last cell that was received from the old end-point, so a minimum of two bits is required for the cyclic counter; in practice more bits may be used to make the system more flexible for other uses.

6.2.7 Call Establishment

The protocol as described so far has considered only cases where a call has already been established so that connexion-identifiers and footprint addresses have been known to both ends. When the call is still being established this information is not known to both ends (or, at least, not known to all of the end-points within a footprint), so complications exist if full mobility is required at this time. The basic principles of the call establishment protocol are shown in plate IV: it is assumed that the remote mobile has been tracked-down to a footprint by interrogation of the H.L.R., and, if necessary, by a paging process to cope with any roaming that has taken place since the last registration. A connexion-request message is transmitted by the end-point associated with the initiating mobile to the footprint identified as being the last known position of the called mobile. It is quite possible that in the time between the paging process completing and the connexion-request message arriving the mobile will have moved from the centre to the periphery of a footprint; but since the connexion-request message is multicast in the same manner as all other messages establishment of the connexion is still possible, the response coming from the end-point in contact with the mobile as opposed to the end-point at the centre of the footprint.

The connexion-identifier to be used when sending cells to the remote mobile is generated on receipt of the connexion-request message. Because the algorithm used to do this is manipulating a dynamic set of identifiers (other calls are setting up, handing off and clearing down simultaneously), and because the mobile may not be at the centre of the footprint addressed by the connexion-request message, it is not possible to generate the number automatically at the other end-points in

the remote footprint. Instead, a message is sent by the remote mobile to its own footprint informing it that a new call has been accepted and includes the relevant connexion information. A connexion-acknowledgement message is sent back to the initiating footprint and this contains sufficient information to configure the end-points in the periphery of the local footprint.

Contingencies covering roaming of the remote mobile during call-setup have been covered above, but those covering roaming of the initiating mobile present greater problems and these fall into two categories. If the mobile wishes to hand-off before the connexion has been established then the hand-off message cannot be generated by the new end-point since the connexion-identifier for the remote end is not yet known. Further, at this stage the end-point in the periphery of the local footprint does not know the connexion-identifier for its own end, since it gets this information in the connexion-acknowledgement message; consequently it is unable to accept the mobile in any case. There are two alternatives that allow this problem to be overcome whilst still permitting roaming during call-setup: the first involves sending a message from the end-point associated with the initiating mobile to the local footprint providing the local connexion information; the second requires that mobiles setting up a call be permitted a tentative hand-off without the local connexion-identifier being recognized. The first option provides a more robust approach, but suffers from the fact that there is still a period, from the start of the setup until the local-connexion message arrives, during which a hand-off is still not possible, and, further, that this period of time does not have a known bound. The best approach, therefore, would appear to be tentative hand-offs; the hand-off message would not be generated until the connexion-acknowledgement message had been received but would then be handled in the usual way. The process limits the mobile to just one hand-off during the call setup, but this should be adequate. Roaming of mobiles during setup was not implemented on the simulator.

6.2.8 Call Disconnexion

Disconnecting a call when it terminates is not a difficult problem. Only the end that initiated the call in the first place is permitted to generate a disconnexion message, the remote end may generate a request for this message to be sent if it should require disconnexion; this simplifies the interactions with the hand-off mechanisms and eliminates entirely any situation where multiple disconnect messages can be passing through the network. Disconnexion is performed without reference to the mobiles (the two processes can be entirely separated); and this makes it impossible for a hand-off sequence to be initiated once the disconnexion procedure has started. If the remote mobile starts a hand-off and an acknowledgement has not been received when a disconnect message arrives, then no acknowledgement will ever be received since the local mobile will simply ignore the hand-off signal: in this way the periphery of the new remote footprint is never established and, therefore, the cleaning up process is much simpler.

6.3 Simulator Modifications

To implement the mobility protocol on the A.T.M. Network Simulator several modifications had to be made, the most complex of which were associated with the traffic-generator modules; previously these were synchronized only with the node into which they fed cells, but now needed connexions to other traffic generators to enable hand-offs to take place. The node models were modified to provide the broadcasting and multicasting capabilities that had previously been omitted from the initial model.

6.3.1 Communication Between Traffic Generators

In a fixed-terminal network, traffic generators in a distributed simulation model can communicate adequately using the signalling messages that would be used in

the real network. With mobile terminals that are permitted to initiate hand-offs without consultation with the network this is no-longer the case; the signalling messages can only be used to indicate that the mobile has moved and not that it is about to do so. It was not practical to model each mobile station as an individual task in the simulator since this would require the simulator topology to be made dynamically reconfigurable to represent the mobile moving about; instead, each mobile was modelled within the fixed traffic-generators as a data structure attached to the active end-point within the footprint. A hand-off was modelled by copying the data structure across a pseudo-connexion between two adjacent traffic generators (adjacency of generators is defined in the configuration file so that hand-offs can occur).

Unfortunately, the movements of the mobile requires that it can only be connected to one end-point at any one time, i.e. that the change of end-point occurs at a unique point in time; and to implement this fully would require that two adjacent traffic generators be synchronized with zero time difference (i.e. no *look-ahead*). This conflicts with the synchronization policy adopted in chapter 3 where an amount of look-ahead is required to permit the processes to run concurrently. Fortunately, the look-ahead can be achieved by calculating in advance the time at which the hand-off will occur; a copy of the mobile's data structure is then sent to the new active point a fixed time before the hand-off itself takes place. Work by Ling [70, 71] suggests that a reasonable model for hand-offs is that they occur with negative-exponential intervals,[6] and this model was adopted with a small modification: the negative-exponential model has a minimum interval of zero which is not acceptable for the synchronization model; a small, constant, time-interval was added to each hand-off interval so that the traffic-generator tasks could be synchronized. This does have a small impact on the distribution function, but

[6]Ling was considering conventional radio-cells where the coverage area is fairly large and geographic features within the area have only secondary importance. In micro-cellular networks, such as are likely in an A.T.M. environment, cells are likely to be individually tailored to suit geographic features (such as a road), making it likely that the negative-exponential model will be a poor one. Despite this, the model was adopted for use within the simulator; all that is required for testing the protocol is for a sufficient density of random hand-offs to occur so that the signalling mechanism can be stressed to its utmost.

since the interval is extremely small in relation to the mean handoff-interval the effect is negligible. A more important consequence of using this technique is that the exact state of the call at the time of the hand-off cannot be passed across the link; and this means that it is difficult to establish whether or not cells from the data stream are lost or duplicated during the hand-off phase. The information can, however, be calculated retrospectively by sending a second message to the new traffic generator at the time that the hand-off actually occurs and by comparing this with the state as it was perceived by the new traffic-generator at the time of the hand-off.

6.3.2 Multicasting Using the Orwell Protocol

Modifications to the Orwell protocol were straight forward. The broadcast capability was implemented, allowing cells to be delivered to all of the nodes on a ring. Multicasting was implemented using special routeing and duplicating tables: cells to be multicast were broadcast round the ring; each node copied the cell to its output link if there was an entry in the multicasting table corresponding to that node and the destination in the cell's A.T.M. header; non-legal values in the multicasting table were indicated with -1. The decision as to whether a cell should be echoed back to the link upon which it was received could be implemented using the Receive Own Broadcast feature of Orwell; in the simulator it was implemented using a flag at the Media Access layer that over-rode the entry in the multicasting table.

6.4 A Practical Implementation of the Protocol

The basic protocol, as already described, was implemented on the simulator. For initial testing hand-offs were restricted to just one end at a time: first the call

initiator; and then the call acceptor. Once each of these were working satisfactorily it was possible to enable both to do this simultaneously. Several problems were rapidly identified with the protocol once both ends were made mobile and these were generally detected by the simulator aborting a run with a diagnostic message. It was not possible to enable the full set of diagnostic messages all of the time since, firstly, the simulator runs much more slowly when producing diagnostics (all of them have to be handled by a single processor and eventually written to either the screen or disk), and secondly the shear volume makes such a brute-force approach impractical; diagnostics are generated at a rate of about 1 Mbyte per minute of real time, and the simulator runs for about thirty minutes for each second of simulated time. Instead, after a problem run has been isolated the time of the error is noted and the simulator re-run with the same set of random-number seeds; shortly before the error occurred the full diagnostics are enabled for the areas of interest, a snap-shot of the status of the simulator can then be taken for analysis. From the results obtained it was possible to establish what the simulator had been doing at the time and, hence, the reason for the protocol failing. Errors usually manifested themselves in one of two ways: different end-points within a footprint having a different understanding about the status of a call; and one end using an incorrect connexion-identifier to that assigned by the other. Individual defects uncovered, and possible solutions, are described in the following sections.

6.4.1 Loss of Hand-off Signal Due to Jitter

Section 6.2.6 noted that it was possible for cells to get lost during a hand-off, and that special measures are required to overcome this. A similar problem applies to the hand-off signals and is shown in plate V: a handoff-request signal from the local mobile can be lost if the remote mobile initiates its own hand-off sequence at the time that the incoming hand-off signal arrives. If the signal arrives at the new, remote, active point just before the hand-off takes place then it will be ignored; similarly if it arrives at the old, remote, active point shortly after the hand-off

takes place then it will be ignored there also. Unlike the data cells, copies cannot be taken and forwarded to the mobile upon hand-off since information about roaming of the remote mobile is handled entirely within the network-endpoints.

The partial solution to the problem involves delaying the old end-point from becoming passive so that it continues to respond to incoming handoff-request messages even after a hand-off has taken place. This will lead to duplicate handoff-acknowledgement messages being generated when just one would have been generated before; but, as will be seen in the next section, this can happen anyway so the full solution involves no further modifications. The old end-point now finally becomes passive on receipt of the handoff-acknowledgement message.

6.4.2 Duplicate Acknowledgements Generated in Response to Hand-off

If the jitter delays that caused signal loss in the above case are reversed then, instead of failing to generate the acknowledgement signal, the acknowledgement is generated by two end-points. The conditions required for this to occur are that the handoff-request signal from the local footprint arrives at the old end-point in the remote footprint shortly before the hand-off takes place (i.e. while the end-point is still active) and this causes an acknowledgement to be generated; if the request arrives at the new end-point shortly after the hand-off has taken place then this too is now active and a second acknowledgement is generated; the signal-flow is shown in plate VI. The two acknowledgement signals contain differing information in the 'remote' fields, the one coming from the old footprint containing details about the connexion prior to hand-off, the other containing the details after hand-off.

Since the two acknowledgement signals are generated almost simultaneously, but travel by different routes back to the local footprint (they come from different places), it is impossible to pre-determine their order of arrival: yet their order is critical to correct interpretation. If the order of arrival of the signals is the same as

shown in plate VI then the signals are all interpreted correctly, the second handoff-acknowledgement signal being ignored. If, however, the acknowledgement signal from the old remote end-point arrives after the handoff-request signal from the remote, or even worse, after the acknowledgement signal from the new remote end-point (plate VII) then the outdated connexion-information that it contains will overwrite the correct remote connexion-information leaving the tables corrupted: it is essential, therefore, that the acknowledgement from the old end-point be ignored if it arrives after either of the signals from the new end-point.

Unfortunately, the original signalling protocol contained no information to make it possible to detect situations such as this, so new fields were necessary: it is not possible to base the acceptance decision on the 'previous value' since this might cause rejection of legal hand-offs that were back to the previous footprint. The new fields in the protocol are simply counters (2-bit, or longer, cyclic counters can be used) that count the number of local and remote hand-offs that have occurred: each end-point within the footprints keeps track of both the local and remote counters and increments the local counter by one when a hand-off to that end-point occurs. When a call is established the initial value of each counter is set to zero, and a particular value of the local counter remains associated with the connexion-identifier whilst the identifier is associated with that call. When signals arrive from the other mobile a simple check will show whether the remote-count indicated in the message is less than the last remote-count known: if it is, then provided that the signal is a hand-off acknowledgement it can be ignored; otherwise it should be flagged as erroneous. A signal with a higher remote-count than previously known indicates that a hand-off has taken place at the remote end, even if the signalling has yet to catch up.

6.4.3 Remote's Handoff-request Arrives Before Acknowledgement of Local Hand-off

This problem can occur when the remote mobile performs a hand-off shortly after receipt of a handoff-request message from the local mobile; it affects end-points

in the periphery of the new footprint, plate VIII. The handoff-request message from the local mobile is handled correctly by the old remote end-point which is still active when the signal is received, and an acknowledgement is generated; shortly afterwards a hand-off occurs and a handoff-request message is generated and directed towards the *new* local footprint; because the two messages take different routes then the handoff-request message from the remote footprint can arrive before the handoff-acknowledgement message. There are two results of this: at the active end-point in the local footprint the handoff-request message containing new data arrives before the acknowledgement containing the outdated data, but this can be resolved in the same way as in the previous section; and at the passive end-points in the periphery of the footprint, the handoff-request message from the remote mobile contains a connexion-identifier for which the end-point has not received an allocation instruction (the instruction will arrive in the acknowledgement message), and when it does arrive it will also contain outdated information. To overcome this problem the end-point needs to be permitted to make a temporary allocation of the designated connexion-identifier subject to confirmation by the pending acknowledgement; the acknowledgement should only be used to confirm that the allocation should have taken place, the information regarding the remote connexion-identifier is outdated by the time it arrives, but this can be detected as before.

6.4.4 Arrival of Handoff-request for a Session Just Deleted

When the network is heavily loaded then delays and jitter effects can become significant; signals taking different routes can become quite widely separated in arrival time and a scenario such as that shown in plate IX can occur. In this case a local hand-off occurs and generates a handoff-request message to the remote footprint. This message arrives at the remote footprint very shortly after a hand-off has occurred there also, but before the old end-point has been passivated.

The old remote end-point generates an acknowledgement that manages to pass through the network significantly more quickly than the handoff-request message coming from the new end-point; the former message has time to be processed by the local mobile and for a footprint-delete message to be issued, and acted upon, by the old local footprint before the handoff-request message arrives. When the handoff-request message finally does arrive from the remote mobile the situation is indistinguishable from that described in the previous section: a request has been received designating a local connexion-identifier that has not been allocated; only the end-point at the centre of the footprint is able to tell that the request is a delayed one, but the data associations to the new connexion-identifier will probably have been cleared by this time. The solution to this requires that old connexion identifiers be given some persistence after a deletion message has been received so that any stragglers are given time to be flushed from the network. The time period need only be a few milliseconds so the effects on the number of available identifiers should not be significant. This is the only time that it was found to be necessary to implement some form of time-out into the basic protocol.

6.5 Summary

The mobility protocol was sucessfully implemented on the simulator, which made posible rapid identification of a series of potential weaknesses; as each of these was diagnosed a work-around was established and implemented. The use of simulation made it possible to test the protocol with scenarios that had not previously been anticipated. The use of distributed simulation enhanced this approach by making it possible to generate results much more quickly than would otherwise have been possible; this in turn makes it less likely that protocol errors have remained undetected.

The fact that the simulator has been seen to successfully complete several simulations is not in itself proof that the protocol is totally free of errors; it is always possible that a particular sequence of events could still lead to an error;

and since the simulator generates events in a random manner it is impossible to guarantee that all such sequences have been covered. All the errors that were detected occurred when both ends were simultaneously in the process of handing off from one end point to another; and the hand-off rate for the simulator was set at a level that was significantly higher than would be seen in a real network. It would seem likely, therefore, that the protocol described here is sufficiently robust for it to be used in a real network.

Chapter 7

Conclusions and Suggestions for Further Work

The work described in this thesis can be broadly divided into three main categories and this chapter will, similarly, summarize them separately. The first section considers distributed simulation and the transputer-based simulator that was designed and implemented as part of this project. Any programming task as large as this always leaves many avenues of interest unexplored; some of these, particularly those related to obtaining better performance, which was only of secondary interest to the main project, are highlighted for further study. The second section considers broadband techniques and the Orwell protocol, whose novel load-control algorithm makes it an extremely interesting topic of study for use in networks with distributed control algorithms. The attempt to find a simulation model for the protocol that used less processing time than the current one was not particularly successful, but remains of great interest even when parallel simulation is being employed. Finally, the mobility protocol is considered. Despite its successful implementation on the simulator, some aspects related to the hand-off make it a little cumbersome and, therefore, potentially fragile: an alteration to the approach is suggested that should remove these frailties and make the whole system significantly easier to implement; at the same time, it should also ease the implementation of network-management and accounting procedures related to the mobile protocol.

7.1 Distributed Simulation and the A.T.M. Network Simulator

Simulation has become an essential weapon in the arsenal of the telecommunications performance engineer; but the dramatic increase in both the physical size of, and the load carried by, networks has meant that conventional simulation technology is finding it hard to cope whilst keeping the computing cost within acceptable bounds. Multi-processor simulators have been proposed that use an array of smaller, cheaper, microprocessors to attack a large simulation in unison to achieve results faster and at significantly lower cost. Several basic decomposition methods have been proposed in the literature and some of the most common were discussed. By far the most popular approach is to break the system being modelled into separate sub-processes and to simulate each of these on a different processor; the processors communicate with each other, passing information about the tasks being processed and the simulation time they have reached: simulation techniques based on this approach are generally termed distributed simulation. An important requirement of the approach is that causality between events occurring within the simulator is maintained; a whole spectrum of approaches aimed at ensuring this have been proposed, but all lie within two basic extremes. At one end of the spectrum, each processor is only permitted to proceed when it can guarantee that the causality of the simulation can be preserved; this approach is generally termed conservative distributed simulation: at the other end, each processor carries on regardless until an error is actually detected, whereupon it restores the local simulation to a previously saved state and restarts from there; not surprisingly this approach is known as optimistic distributed simulation. Conservative algorithms are usually the easiest to implement, but unfortunately normally tend to give poorer performance than their optimistic counterparts; in their favour, however, they can usually run with less memory, are easier to debug, and, with sufficiently good look-ahead, can give quite reasonable performance.

In discrete-event simulators written prior to the early nineteen-eighties very little consideration was given to how the event set should be maintained: the 'obvious' approach was to maintain a linear list sorted in order of increasing time. As the size of simulations grew, the length of the list made it impractical for the use of such a *naïve* approach to be continued and more sophisticated list-management techniques were adopted. Consideration of the partitioning technique used when a distributed simulator is written shows that, when considering the processing requirements for the event set alone, super-unitary speed-up is possible; this contradicts an earlier claim in the literature.

It has been found from experience that single-processor computers are only able to simulate in great detail, and in a realistic time-frame, single nodes of very high-speed networks. A simulation package, the A.T.M. Network Simulator, was written to enable high-speed simulation of Broadband I.S.D.N. networks at the cell level; the package runs on a reconfigurable array of Inmos T800 transputers with each processor simulating one node in the network. The code for the simulator was written in 'C' and was developed in a hierarchical manner. At the bottom layer was a multiplexor system that was both deadlock and livelock free; and which provided a virtual interconnexion network between each task running in the simulation. A packetizer layer was used to make efficient use of the communications bandwidth of the network. Finally, a synchronization layer was used to perform a form of conservative synchronization between the tasks: the layer was proved to be deadlock free and to maintain causality relationships between the tasks at all times. The performance of the simulator was found to be very good, giving near-ideal speed-up over a large range of loads when identical copies of the code were run on first a single processor and then on the multi-processor network.

The simulator uses a 'backing-off' approach to generating NULL messages. When such a message has been generated the simulator reschedules the generation event to re-run one look-ahead period later on. If, on the next occasion that the event runs it is found that in the meantime a 'real' message has been

transmitted then the event can be backed off until one look-ahead period after that message was transmitted, and so on. When there is a large population of real messages this means that the number NULL messages required falls almost to zero. However, when the real-message density is low then if the last real message to be transmitted was sent only very shortly after the last NULL message then backing off the NULL message for just a short period of time may be less efficient than sending a NULL message immediately and backing off for longer. Within the bounds of the synchronization approach (that at least one message should be transmitted each look-ahead period), any NULL message generation algorithm is acceptable: perhaps the best approach might be an adaptive one that bases its decision as to whether it is better to back off or to send a non-essential message; the trade off is between work required to reschedule the event and work required to transmit a message; greater concurrency might also be achieved by sending more messages since it is then less likely that the remote process will block.

Another feature of the simulator that still has scope for expansion is the traffic generator model. At present only voice and mobile traffic are generated (with some signalling), and both of these are fixed bit-rate services; variable bit-rate services such as computer data and compressed video would greatly enhance the adaptability of the simulator. Connexion management within the network is another area where the current simulator implements only minimal functionality: since the use for the simulator in this project was to study end-to-end protocols over an A.T.M. environment, this was not a major limitation; but for studies of the network itself, more development in this area would be required.

The parallelism extracted from the systems simulated exceeded all expectations, giving near-ideal speed-up over a very wide range of loads. This can probably be attributed, at least in part, to the low efficiency of the model used for the Orwell protocol: much time is spent by each ring rotating slots for which there is little or no traffic; this creates a large number of events between each synchronization and is sufficient to keep a processor busy for a large part of the time.

7.2 Asynchronous Transfer Mode and Orwell

A novel switching technique is being actively developed throughout the world in an attempt to establish an international standard for broadband communications using public networks. The technique is known as Asynchronous Transfer Mode (A.T.M.) because each data packet, or cell, contains sufficient information in its header to enable it to be independently routed through a network. Unlike conventional circuit-switched networks connexions can be allocated for any amount of bandwidth that is required, rather than as multiples of a basic rate. Further, statistical multiplexing can be used to merge variable bit-rate connexions together so that it is not necessary to allocate the peak bandwidth requirement to each.

Conventional packet-switched networks have often used statistical multiplexing to achieve reductions of the bandwidth requirement within the network, but for A.T.M. conventional store-and-forward networks are too cumbersome, and indeed their link-level protocols are no longer required for high-reliability fibre-optic links. New switching architectures are also required to support the very high throughputs that will be required of A.T.M. networks; traditional packet-switching architectures simply do not have the capacity and throughput to support them. A new protocol that has been proposed for use in high-speed local and metropolitan area networks, and for use in low-capacity A.T.M. exchanges, is the Orwell ring protocol; this uses a rotating slot principle as its access mechanism. Earlier slotted ring protocols had fallen into disfavour because they commonly had high header-to-data ratios and poor load balancing capabilities; but Orwell has no worse header-to-data ratio than any other exchange protocol as this is defined by the A.T.M. standard; and it provides a novel access control mechanism to guarantee bandwidth in accordance with the needs of each node on the ring. Simulation times for ring protocols tend to be high, since much time must be spent simulating the slot rotation action; attempts were made to reduce simulation times required to simulate the Orwell protocol by substituting the slot's seeking action with a simpler system that requires less computation. Unfortunately substantial performance gains could not be achieved and in the process the

queueing characteristics of the ring were slightly changed. Two distinct phases were found to be involved with the switching of each cell: during the first phase an empty slot is seeking traffic that it can carry, and the time for this is dependent on the load offered to the ring; during the second phase a slot has found traffic and is carrying it round the ring to its destination, this is independent of the load offered to the ring and is, typically half a rotation delay.

7.3 Mobility Using A.T.M. Networks

Mobile communication is one of the fastest-growing fields of the expanding telecommunications industry; so it is important that services such as this should be considered when defining the A.T.M. protocols. It has been proposed that mobile communications could be supported in a micro-cellular radio environment by using a footprinting technique: instead of disseminating information about a call to just the base stations handling each end of the call, all the information (including the incoming data stream) is duplicated within the network and sent also to base-stations that surround the one that is currently active. Handing a call from one base-station to another then no-longer requires a new connexion to be established in the time between detecting that the current one is fading and total loss of communication: the connexion information already exists and simply needs updating. Once a hand-off has taken place, a new footprint is established around the new location of the mobile so that the next hand-off can also be in any direction; this is a far less time-critical process than exists in current mobile protocols. Further, such an approach can also be extended to other mobility environments, for example when a user is communicating with a single computer using several terminals: each terminal needs to receive data from the main computer, but the user may be moving rapidly from one to another; a footprint multicasting technique can be used here also.

The proposed protocol was implemented on the A.T.M. network simulator in an attempt to test the signalling capabilities in a realistic environment, where

signalling delays along divers paths are all different and can lead to messages arriving out of order. It was found that the basic hand-off protocol worked reliably when just one end was moving at a time; and adequately when both ends moved simultaneously. Problems occurred, however, when the signal indicating that a mobile had handed off arrived at the remote mobile at approximately the same time that it too was initiating its own hand-off sequence; depending upon the exact sequence of events each end would determine a different state of the connexion and the effect was in some respects similar to crossed lines in conventional telephony (a problem now largely eradicated). These problems could only be overcome with some difficulty and at the expense of burdening the protocol with additional clauses.

One of the problems with mobile communications is that the customers keep moving! This makes network management (and particularly call charging calculations) especially difficult: one way of overcoming this problem has been to introduce the concept of an anchor node; this is selected when the call is established, and remains fixed throughout the duration of the call. The current proposals for implementing the new mobility protocol have not, so far, implemented the concept of an anchor node so one mobile station has been communicating directly with another and there has been no reference point within the network. Some of the difficulties found with the protocol, however, could almost certainly be eliminated by using an anchor node in the communications path: the protocol proved to be highly reliable when only one end was permitted to move at a time; so introducing an anchor node, and having the mobiles communicate indirectly with each other by using it, would make movement of the 'remote' mobile totally transparent to movement of the 'local' one. The anchor node, in effect, acts as a fire wall to shield the movements of the mobiles from each other. This would require that the base stations generate cells addressed to the anchor nodes and would further require the anchor node to process cells at the A.T.M. adaptation layer (in a similar way to zone boundaries in the A.T.M. zone concept); but the advantages would seem to make further investigation of this approach worthwhile.

Finally, the protocol still needs to be tested under conditions where cells are being dropped by the network causing signals to be missed; the anchor node may also be beneficial here, shielding the mobiles from each other and simplifying the recovery mechanism by reducing the number of possible state-transitions in the system.

Bibliography

[1] K. M. Chandy, V. Holmes, and J. Misra, "Distributed Simulation of Networks," *Computer Networks*, vol. 3, pp. 105–113, 1979.

[2] K. M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 440–452, September 1979.

[3] R. E. Bryant, "Simulation of Packet Communication Architecture Computer Systems," tech. rep., M.I.T., Cambridge, MA, November 1977. MIT,LCS,TR-188.

[4] C. A. R. Hoare, *Communicating Sequential Processes.* Prentice Hall International, 1985. I.S.B.N. 0-13-153289-8.

[5] J. Misra, "Distributed Discrete-Event Simulation," *Computing Surveys*, vol. 18, pp. 39–65, March 1986.

[6] R. M. Fujimoto, "Lookahead in Parallel Discrete Event Simulation," in *Proceedings of the IEEE International Conference on Parallel Processing*, vol. 3, pp. 34–41, 1988.

[7] R. M. Fujimoto, "Performance Measurements of Distributed Simulation Strategies," in *Distributed Simulation*, pp. 14–20, February 1988.

[8] D. A. Reed, A. D. Malony, and B. D. McCredie, "Parallel Discrete Event Simulation Using Shared Memory," *IEEE Transactions on Software Engineering*, vol. 14, pp. 541–553, April 1988.

[9] D. A. Reed and A. D. Malony, "Parallel Discrete Event Simulation: The Chandy-Misra Approach," in *Distributed Simulation*, pp. 8–13, February 1988.

[10] D. M. Nicol, "High Performance Parallelized Discrete Event Simulation of Stochastic Queueing Networks," in *Proceedings of the 1988 Winter Simulation Conference*, pp. 306–314, 1988.

[11] W. Cai and S. J. Turner, "An algorithm for Distributed Discrete-event Simulation — The "Carrier Null Message" Approach," in *Distributed Simulation*, pp. 3–8, January 1990.

[12] W. K. Su and C. L. Seitz, "Variants of the Chandy-Misra-Bryant Distributed Discrete-event Simulation Algorithm," in *Distributed Simulation*, pp. 38–43, March 1989.

[13] R. C. De Vries, "Reducing Null Messages in Misra's Distributed Discrete Event Simulation Method," *IEEE Transactions on Software Engineering*, vol. 16, pp. 82–91, January 1990.

[14] K. M. Chandy and R. Sherman, "The Conditional Event Approach to Distributed Simulation," in *Distributed Simulation*, pp. 93–99, March 1989.

[15] M. Abrams, "The Object Library for Parallel Simulation," in *Proceedings of the 1988 Winter Simulation Conference*, pp. 210–219, 1988.

[16] M. Abrams, "A Common Programming Structure for the Bryant-Chandy-Misra, Time-Warp, and Sequential Simulators," in *Proceedings of the 1989 Winter Simulation Conference*, pp. 661–670, December 1989.

[17] D. M. Nicol, C. C. Micheal, and P. Inouye, "Efficient Aggregation of Multiple LPs in Distributed Memory Parallel Simulations," in *Proceedings of the 1989 Winter Simulation Conference*, pp. 680–685, 1989.

[18] B. R. Preiss, W. M. Loucks, and V. C. Hamacher, "A Unified Modelling Methodology for Performance Evaluation of Distributed Discrete Event Simulation Mechanisms," in *Proceedings of the 1988 Winter Simulation Conference*, pp. 315–324, 1988.

[19] B. R. Preiss, "The Yaddes Distributed Discrete Event Simulation Specification Language and Execution Environments," in *Distributed Simulation*, pp. 139–144, March 1989.

[20] D. R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 404–425, July 1985.

[21] D. Jefferson and H. Sowizral, "Fast Concurrent Simulation using the Time Warp Mechanism," in *Distributed Simulation*, pp. 63–69, January 1985.

[22] D. Jefferson, B. Beckman, S. Hughes, E. Levy, and T. Litwin, "Implementation of Time Warp on the Caltech Hypercube," in *Distributed Simulation*, pp. 70–75, January 1985.

[23] R. M. Fujimoto, J. J. Tsai, and G. Gopalakrishnan, "The Roll Back Chip: Hardware Support for Distributed Simulation using Time Warp," in *Distributed Simulation*, pp. 81–86, February 1988.

[24] C. A. Buzzell, M. J. Robb, and R. M. Fujimoto, "Modular VME Rollback Hardware for Time Warp," in *Distributed Simulation*, pp. 153–156, January 1990.

[25] S. Bellenot, "Global Virtual Time Algorithms," in *Distributed Simulation*, pp. 122–127, January 1990.

[26] P. L. Reiher, F. Wieland, and D. Jefferson, "Limitation of Optimism in the Time Warp Operating System," in *Proceedings of the 1989 Winter Simulation Conference*, pp. 765–770, 1989.

[27] D. Baezner, G. Lumow, and B. W. Unger, "Sim++: The Transition to Distributed Simulation," in *Distributed Simulation*, pp. 211–218, January 1990.

[28] P. F. Reynolds Jr, "A Spectrum of Options for Parallel Simulation," in *Proceedings of the 1988 Winter Simulation Conference*, pp. 325–332, 1988.

[29] R. L. Gimarc, "Distributed Simulation using Hierarchical Rollback," in *Proceedings of the 1989 Winter Simulation Conference*, pp. 621–629, December 1989.

[30] B. Lubachevsky, A. Shwartz, and A. Weiss, "Rollback Sometimes Works ... If Filtered (Abstract)," in *Proceedings of the 1989 Winter Simulation Conference*, pp. 630–639, December 1989.

[31] B. D. Lubachevsky, "Bounded Lag Distributed Discrete Event Simulation," in *Distributed Simulation*, pp. 183–191, February 1988.

[32] R. J. Lipton and D. W. Mizell, "Time Warp vs. Chandy-Misra: A Worst-case Comparison," in *Distributed Simulation*, pp. 137–143, January 1990.

[33] S. M. Swope and R. M. Fujimoto, "Optimal Performance of Distributed Simulation Programs," in *Proceedings of the 1987 Winter Simulation Conference*, pp. 612–617, December 1987.

[34] D. W. Jones, "Concurrent Simulation: An Alternative to Distributed Simulation," in *Proceedings of the 1986 Winter Simulation Conference*, pp. 417–423, December 1986.

[35] D. W. Jones, Chien-Chun Chou, D. Renk, and S. C. Bruell, "Experience with Concurrent Simulation," in *Proceedings of the 1989 Winter Simulation Conference*, pp. 756–764, 1989.

[36] C. Hughes, U. Chandra, and S. V. Sheppard, "Two Implementations of a Concurrent Simulation Environment," in *Proceedings of the 1987 Winter Simulation Conference*, pp. 618–623, December 1987.

[37] P. F. Reynolds Jr, "Heterogenous Distributed Simulation," in *Proceedings of the 1988 Winter Simulation Conference*, pp. 206–209, 1988.

[38] P. Heidelburger, "Statistical Analysis of Parallel Simulations," in *Proceedings of the 1986 Winter Simulation Conference*, pp. 290–295, December 1986.

[39] P. W. Glynn and P. Heidelburger, "Analysis of Parallel Replicated Simulations Under a Completion Time Constraint," *A.C.M. Transactions on Modeling and Computer Simulation*, vol. 1, pp. 3–23, January 1991.

[40] N. J. Bailey, *On the Synthesis and Processing of High Quality Audio Signals by Parallel Computers.* PhD thesis, University of Durham, October 1991.

[41] N. J. Bailey, A. Purvis, I. W. Bowler, and P. D. Manning, "An Highly Parallel Architecture for Real-time Music Synthesis and Digital Signal Processing Application," in *Proceedings of the International Music Conference, Glasgow*, 1990.

[42] N. J. Bailey, A. Purvis, I. W. Bowler, and P. D. Manning, "Some Observations on Hierarchichal, Multiple-Intruction-Multiple-Data Computers," in *Proceedings of Euromicro, Vienna*, September 1991.

[43] D. W. Jones, "An Empirical Comparison of Priority-Queue and Event-Set Implementations," *Communications of the ACM*, vol. 29, pp. 300–311, April 1986.

[44] J. H. Blackstone Jr, G. L. Hogg, and D. T. Phillips, "A Two-List Synchronization Procedure for Discrete Event Simulation," *Communications of the ACM*, vol. 24, pp. 825–829, December 1981.

[45] D. D. Sleator and R. E. Tarjan, "Self-Adjusting Binary Search Trees," *Journal of the Association for Computing Machinery*, vol. 32, pp. 652–686, July 1985.

[46] D. P. Helmbold and C. E. McDowell, "Modeling Speedup (n) Greater than n," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 250–256, April 1990.

[47] S. J. Nichols, *Simulation and Analysis of Adaptive Routeing and Flow Control in Wide Area Communication Networks.* PhD thesis, University of Durham, March 1990.

[48] R. T. Clarke, S. J. Nichols, and P. Mars, "Transputer-based Simulation Tool for Performance Evaluation of Wide Area Telecommunications Networks," *Microprocessors and Microsystems*, vol. 13, pp. 173–178, April 1989. Special issue on transputer applications.

[49] S. Toueg and J. D. Ullman, "Deadlock-Free Packet Switching Networks," in *Proceedings of the ACM Symposium on the Theory of Computing, Atlanta, Georgia*, pp. 89–98, May 1979.

[50] L. Gould, I. Bowler, and A. Purvis, "Real-Time, Multi-Channel Digital Filtering on the Transputer," in *Proceedings of the 1989 International Symposium on Computer Architecture and Digital Signal Processing*, 1989.

[51] 3L Ltd, *Parallel C User Guide, Version 2.1.* Peel House, Ladywell, Livingston EH54 6AG, Scotland, 1989.

[52] M. De Prycker, "Data Communication in an ATM Network," *Electrical Communication*, vol. 62, no. 3/4, pp. 333–337, 1988.

[53] C.C.I.T.T. S.G.XVIII, Brasilia, *Load Increase Under Link-By-Link and End-To-End Error Recovery*, February 1987. Contribution 987 (U.S.A.).

[54] J. Dupraz and M. De Prycker, "Principles and Benefits of the Asynchronous Transfer Mode," *Electrical Communication*, vol. 64, no. 2/3, pp. 116–123, 1990.

[55] G. Foster and J. L. Adams, "The ATM Zone Concept," in *IEEE Globecom 1988*, vol. 2, pp. 21.4.1–21.4.3, 1988.

[56] A. Huang and S. Knauer, "Starlite: A Wideband Digital Switch," in *IEEE Globecom Vol. 1 of 3, Atlanta, Georgia*, pp. 121–125, November 1984.

[57] J. Chauhan, T. King, and A. C. Micallef, *Specification of the Orwell Protocol.* British Telecom Research Laboratories, Martlesham Heath, Ipswich, Suffolk, UK. IP5 7RE, May 1990. Revision C.1(05/90).

[58] J. Appleton, "Traffic Shaping in Asynchronous Transfer Mode Networks," in *Proceedings of the Eighth U.K. Teletraffic Symposium*, pp. 19/1–19/4, April 1991.

[59] J. R. Chen and P. Mars, "Adaptive ATM Call Access Control Using Learning Algorithms," in *Proceedings of the Eighth U.K. Teletraffic Symposium*, pp. 18/1–18/5, April 1991.

[60] R. M. Falconer, J. L. Adams, and G. M. Walley, "A Simulation Study of the Cambridge Ring with Voice Traffic," *British Telecom Technology Journal*, vol. 3, April 1985.

[61] J. L. Adams and R. M. Falconer, "Orwell: A Protocol for Carrying Integrated Services on a Digital Communictions Ring," *Electronics Letters*, vol. 20, pp. 970–971, November 1984.

[62] R. M. Falconer and J. L. Adams, "Orwell: A Protocol for an Integrated Services Local Network," *British Telecom Technology Journal*, vol. 3, October 1985.

[63] S. A. Johnson, "Description of Orwell System Simulation Model," tech. rep., British Telecom Research Laboratories, Martlesham Heath, Ipswich, Suffolk, U.K. IP5 7RE, January 1988.

[64] D. J. Miller, "Orwell Simulation User Guide," Tech. Rep. 379.2.2, British Telecom Research Laboratories, Martlesham Heath, Ipswich, Suffolk, U.K. IP5 7RE, July 1989.

[65] I. D. Gallagher, "Multi-Service Networks," *British Telecom Technology Journal*, vol. 4, pp. 43–49, January 1986.

[66] M. Littlewood, I. D. Gallagher, and J. L. Adams, "Evolution Toward an ATD Multi-Service Network," *British Telecom Technology Journal*, vol. 5, April 1987.

[67] J. L. Adams, "Support of ATM on a Passive Optical Local Access System," in *Workshop on Asynchronous Transfer Mode*, (CICG, Geneva), pp. 6.4.1–6.4.9, June 1988.

[68] B. M. McGeeney, "Performance of an Integrated Services ATM Protocol over a Broadband Passive Optical Network," in *Proceedings of the Sixth UK Teletraffic Symposium*, pp. 12/1–12/8, May 1989.

[69] W. C. Y. Lee, *Mobile Cellular Telecommunication Systems.* McGraw-Hill, 1989. I.S.B.N. 0-07-037030-3.

[70] Y. K. Ling, "Second Technical Report of "Wide Area Roaming in Cellular Mobile Radio"," tech. rep., University of Essex, Department of Electronic Systems Engineering, April 1989.

[71] Y. K. Ling, "Third Technical Report of "Wide Area Roaming in Cellular Mobile Radio"," tech. rep., University of Essex, Department of Electronic Systems Engineering, December 1989.

Appendix A

Published Papers

The following papers have been published, or have been submitted for future publication, as a result of this work:

- R. W. Earnshaw, A. Titchmarsh and P. Mars, "Design and Implementation of a Packet-switched Network Simulator," in *Proceedings of the Sixth I.E.E. U.K. Teletraffic Symposium*, May 1989.

- R. W. Earnshaw and P. Mars, "Simulation of A.T.M. Networks on Transputer Arrays," in *Proceedings of the Seventh I.E.E. U.K. Teletraffic Symposium*, pp. 1/1–1/5, April 1990.

- R. W. Earnshaw and P. Mars, "Footprints for Mobile Communications," in *Proceedings of the Eighth I.E.E. U.K. Teletraffic Symposium*, pp. 22/1–22/5, April 1991.

- R. W. Earnshaw and A. Hind, "Parallel Simulation of Asynchronous Transfer Mode Networks," to be presented at *The Fourth I.E.E. Bangor Communications Symposium, University of Wales, Bangor*, May 1992.

- R. W. Earnshaw and A. Hind, "A Parallel Simulator for Performance Modelling of Broadband Telecommunications Networks," submitted to *the I.E.E.E. Winter Simulation Conference, Arlington, Virginia*, December 1992.

Appendix B

Source Listing of Mobile Functions

This appendix contains the source listings for the mobility functions on the A.T.M. network simulator. For reasons of space the whole code for the simulator (about 20,000 lines) cannot be included: a floppy disk is included with the thesis containing source listings for the simulator and several utilities; it is formatted for an I.B.M. P.C.-A.T..

B.1 Mobile.h

```
/*********************************
 *
 * ATM Network Simulation Package
 *
 * Traffic Generator Model: MOBILE.H
 *
 * Author: R.W.Earnshaw, University of Durham
 *
 * Date: 29th October, 1990
 *
 * Copyright (C) 1990, University of Durham, All rights reserved.
 *
 *##############################
 *
 * MOBILE.H: Structures for mobile calls within a source
 *
 *********************************/

#ifdef MOBILITY
#ifndef STATS_HOPS_LIM
#define STATS_HOPS_LIM 3
#endif /* STATS_HOPS_LIM */
```

```
#define MS_ACCEPT 1
#define MS_FAILED 2
#define MS_CLEAR 3

#define FTPRNT_MASK 0x0000ffffl
#define SESSION_MASK 0xffff0000l
#define SESSION_OFFSET 16

typedef struct mobile_calls MOBILE_CALLS;

typedef struct m_call M_CALL;

typedef struct m_session M_SESSION;

typedef struct ses_header SES_HEADER;

typedef struct ho_chan HO_CHAN;

struct ho_chan
{
    RADIO_CHANNEL *rad_chan;
};

struct ses_header
{
    M_SESSION **sessions;
    HO_CHAN *handoff;
};

struct mobile_calls
{
    STATS_FN hold_func;
    STAT_DIST hold_dist;
    long hold_seed;
    STATS_FN create_func;
    STAT_DIST create_dist;
    long create_seed;
    double sample_rate;
    int setup_mes_len;
    int setup_ack_len;
    int clear_mes_len;
    int clear_ack_len;
    int footprint_len;
    int handoff_len;
    NET_CHANNEL *channel;
    M_CALL *running_calls;
    M_CALL *my_calls;
    D_STATS *d_stats;
    int num_dests;
    SES_HEADER *session_table;
    int max_sessions;
    int *ftprnt_lkup;
    long ftprnt_seed;
```

```
    int num_ftprnt_cells;
    STATS_FN handoff_func;
    STAT_DIST handoff_dist;
    long handoff_seed;
    TIME del_session_wait;
};

struct m_call
{
    M_CALL *next;
    M_CALL *prev;
    M_CALL *call_next;
    M_CALL *call_prev;
    unsigned long state;
    unsigned long session;
    int master;
    int handoffs_done;
    TIME time_next;
    TIME time_term;
    RADIO_CHANNEL *handoff_on;
    TIME handoff_at;
};

struct m_session
{
    M_SESSION *next;
    M_SESSION *new_session;
    M_SESSION *old_session;
    M_CALL *call;
    FRAME *last_frame;
    int last_frame_size;
    unsigned long my_session_id;
    unsigned long your_session_id;
    int active;
    unsigned long state;
    int handoff_no_txed; /* The tag on my last handoff */
    int handoff_no_rxed; /* The tag on last remote handoff */
};

PUBLIC TIME inform_handoff (M_CALL *call, TIME time);
PUBLIC TIME mobile_call_create (void *data, TIME time);
PUBLIC void ms_connect_confirm (M_CALL *call, int status,
        unsigned long session, TIME time);
PUBLIC void ms_connect_indication (unsigned long session, TIME term, TIME time);
PUBLIC void ms_data_indication (M_CALL *call, FRAME *frame, int size,
        TIME time);
PUBLIC TIME send_mobile_cell (MOBILE_CALLS *m_calls, TIME time);
PUBLIC TIME m_terminate_call (M_CALL *call, TIME time);
PUBLIC void ms_release_indication (M_CALL *call, int reason, TIME time);
PUBLIC void ms_release_confirm (M_CALL *call, int reason, TIME time);
PUBLIC void ms_connect_request (unsigned long address, M_CALL *call, TIME term,
        TIME time);
PUBLIC void ms_connect_response (unsigned long session_no, M_CALL *call,
```

```
        int state, TIME time);
PUBLIC void ms_data_request (unsigned long session_no, FRAME *frame, int size,
        TIME time);
PUBLIC void ms_release_request (unsigned long session_no, int reason,
        TIME time);
PUBLIC void ms_release_response (unsigned long session_no, int status,
        TIME time);
PUBLIC void mobile_setup_req (FRAME *frame, int size, TIME time);
PUBLIC void mobile_setup_ack (FRAME *frame, int size, TIME time);
PUBLIC void mobile_term_req (FRAME *frame, int size, TIME time);
PUBLIC void mobile_term_ack (FRAME *frame, int size, TIME time);
PUBLIC void mobile_footprint (FRAME *frame, int size, TIME time);
PUBLIC void mobile_cell_rx (CELL *cell, TIME time);
#endif /* MOBILITY */
```

B.2 Mobile.c

```
/*****************************
 *
 * ATM Network Simulation Package
 *
 * Traffic Generator Model: MOBILE.C
 *
 * Author: R.W.Earnshaw, University of Durham
 *
 * Date: 10th December, 1990
 *
 * Copyright (C) 1990, University of Durham, All rights reserved.
 *
 *############################
 *
 * MOBILE.C: Mobility functions traffic generator, second attempt.
 *
 *****************************/

#ifndef lint
static char rcsid[] =
"$Id:  MOBILE.C 1.1 91/03/19 10:28:07 erich Exp Locker:  erich $";
#endif /* lint */

/*
 * $Log: MOBILE.C $
 * Revision 1.1 91/03/19 10:28:07 erich
 * Initial revision
 *
 */

#include "../include/atm.h"
#include "../include/timeout.h"
#include "../include/allocate.h"
#include "../include/stats.h"
```

```
#include "../include/io.h"
#include "../include/network.h"
#include "../include/radio.h"
#include "../include/protocol.h"
#include "../include/cell.h"
#include "../include/mac.h"
#include "mobile.h"

#ifdef DEBUG
#define dbg(a) {if (debug & 128) {a}}
#else
#define dbg(a) {}
#endif /* DEBUG */

/* Define a few functions for access to the structures */


/* The following function may be made to do a more elaborate search for a
 * mobile within the network. Currently we assume that this has all been
 * done in the background and simply return an address.
 *
 * Syntax:
 *
 * unsigned locate_mobile (unsigned mobile_id);
 *
 * mobile_id is the mobile's identiy number.
 */

#define locate_mobile(num) ((int) ((num) % m_calls_data.num_dests))

/* Overload status operations */

#define get_status(ent) ((ent)->state)
#define set_status(ent,status) ((ent)->state = (status))

/* Running call information */

#define link_f_call(call,nxtcall) ((call)->call_next = (nxtcall))
#define link_b_call(call,prvcall) ((call)->call_prev = (prvcall))

#define get_f_call(call) ((call)->call_next)
#define get_b_call(call) ((call)->call_prev)

/* Other call information */

#define set_terminate(call,time) ((call)->time_term = (time))
#define get_terminate(call) ((call)->time_term)
#define ismaster(call) ((call)->master)

#define get_generate_time(call) ((call)->time_next)
#define set_next_generate(call,delay) ((call)->time_next += (delay))
#define init_generate(call,time) ((call)->time_next = (time))
```

```
#define generate_mobile_id() (vprng (m_calls_data.hold_seed))

#define get_last_arrival(call) ((call)→last_arrival)
#define set_last_arrival(call,time) ((call)→last_arrival = (time))

#define get_running_calls() (m_calls_data.running_calls)
#define set_running_calls(call) (m_calls_data.running_calls = (call))

#define get_calls_list() (m_calls_data.my_calls)
#define set_calls_list(call) (m_calls_data.my_calls = (call))
#define gen_new_footprint() \
        (m_calls_data.ftprnt_lkup[((int) ((m_calls_data.ftprnt_seed =\
        vprng (m_calls_data.ftprnt_seed)) % m_calls_data.num_ftprnt_cells))])
#define get_footprint(sess_id) ((int)((sess_id) & FTPRNT_MASK))
#define get_session_no(sess_id) \
        ((int)(((sess_id)&SESSION_MASK)≫SESSION_OFFSET))
#define set_session_id(id,ftprnt,no) (*(id) = (ftprnt)|((no)≪SESSION_OFFSET))

#define in_footprint(ftprnt) \
        (m_calls_data.session_table[ftprnt].sessions == NULL ? 0: 1)

#define which_session(ses_id) \
        (get_session_no(ses_id) < m_calls_data.max_sessions ? \
        m_calls_data.session_table[get_footprint(ses_id)].\
                sessions[get_session_no(ses_id)] : NULL)

#define mobile_in_footprint(ft,add) ((ft) == source_id ? 1 : 0)

#define new_call(call) if ((*(call) = m_call_alloc ()) == NULL) \
                        { \
                            root_message ("Unable to create new mobile call");\
                            terminate_sim (); \
                        }

#define new_frame(frame) if ((*(frame) = frame_alloc ()) == NULL) \
                        { \
                            root_message ("Unable to create new frame");\
                            terminate_sim (); \
                        }

#define add_calls_list(call) { \
                                link_forward(call,m_calls_data.my_calls); \
                                link_backward(call, NULL); \
                                if (get_calls_list () ≠ NULL) \
                                    link_backward(get_calls_list (), call);\
                                set_calls_list (call); \
                            }

#define start_timeout(what,when)\
        add_timeout(&what##_timeout, what, ((what)→time_out = (when)))

/*#####################################*/
```

```
/* Mobile level status definitions */

#define M_SETUP_STATE 0x00000001
#define M_RUNNING_STATE 0x00000002
#define M_CLEAR_STATE 0x00000004

/* Session level status definitions */

#define S_SETUP_STATE 0x00000001
#define S_RUNNING_STATE 0x00000002
#define S_CLEAR_STATE 0x00000004
#define S_HANDOFF_F_STATE 0x00000008 /* This session has gone inactive */
                                     /* during handoff */
#define S_HANDOFF_T_STATE 0x00000010 /* This session has gone active */
                                     /* during handoff */
#define S_HANDOFF_E_STATE 0x00000020 /* For recovery purposes */
#define S_DELETE_STATE 0x0000000100 /* Set when the session is about */
                                    /* to expire */
#define S_DUPLICATE_SENT 0x00008000

#define M_SETUP_REQ 1
#define M_SETUP_ACK 2
#define M_CLEAR_REQ 3
#define M_CLEAR_ACK 4
#define M_CREATE_FOOTPRINT 5
#define M_DELETE_FOOTPRINT 6
#define M_HANDOFF_REQ 7

#define SES_FORCE 0
#define SES_GENERATE 1

#define HANDOFF_LEAVING 0
#define HANDOFF_ARRIVING 1

#define NOT_RUNNING 0
#define MOBILE_SETUP 1
#define MOBILE_DATA 2
#define MOBILE_CLEAR 3

#define SETUP_OK 101
#define SETUP_FAILED 102
#define MOBILE_DROPPED 103

#define M_MASTER TRUE
#define M_SLAVE FALSE

/*#####################################*/


PRIVATE void unlink_call (M_CALL *call);
PRIVATE void mobile_call_insert (M_CALL *call, TIME time);
PRIVATE void new_session (M_SESSION **session, int generate,
                                    unsigned long footprint);
```

```
PRIVATE void unlink_session (M_SESSION *session);

PUBLIC TIME release_session (M_SESSION *session, TIME time);

PUBLIC MOBILE_CALLS m_calls_data;

PUBLIC extern int source_id;

PUBLIC int mobile_handoffs = 0;

extern void (*make_header) ();

allocator (PRIVATE, M_CALL, m_call, {})
allocator (PRIVATE, M_SESSION, m_session, {})

/************ Mobile Level (user) calls **************/

PUBLIC TIME mobile_call_create
(
    void *data,
    TIME time
)
{
M_CALL *call;

    if (time > 0.80)
        debug |= 128;
    new_call (&call);
    call->master = M_MASTER;
    call->handoffs_done = 0;
    add_calls_list (call);
    link_f_call (call, NULL);
    link_b_call (call, NULL);
    set_terminate (call, time + distribution (&m_calls_data, hold));
    set_status (call, M_SETUP_STATE);
    dbg (root_message ("MOBILE: Call created");)
    ms_connect_request (generate_mobile_id (), call, get_terminate (call),
            time);
    return (time + distribution (&m_calls_data, create));
}

PUBLIC void ms_connect_confirm
(
    M_CALL *call,
    int status,
    unsigned long session,
    TIME time
)
{
    dbg (root_message ("MOBILE: Received Acknowlegement, Session %x",
            session);)
    switch (status)
    {
```

```
        case MS_FAILED:
            unlink_call (call);
            m_call_free (call);
            break;
        case MS_ACCEPT:
            if (get_status (call) ≠ M_SETUP_STATE)
            {
                root_message ("MOBILE: Not in setup when received ack");
                terminate_sim ();
            }
            call→session = session;
            mobile_call_insert (call, time + CELL_INFO / m_calls_data.sample_rate);
            dbg (root_message ("MOBILE: Setting handoff time");)
            gen_next_handoff (call, time);
            break;
        }
}

PUBLIC void ms_connect_indication
(
    unsigned long session,
    TIME term,
    TIME time
)
{
M_CALL *call;

    dbg (root_message ("MOBILE: Received connect indication.  Session %x",
            session);)
    new_call (&call);
    call→master = M_SLAVE;
    call→handoffs_done = 0;
    call→time_term = term;
    add_calls_list (call);
    call→session = session;
    link_f_call (call, NULL);
    link_b_call (call, NULL);
    set_status (call, M_RUNNING_STATE);
    mobile_call_insert (call, time + CELL_INFO / m_calls_data.sample_rate);
    /* send the response message */
    ms_connect_response (session, call, MS_ACCEPT, time);
    gen_next_handoff (call, time);
}

PUBLIC TIME send_mobile_cell
(
    MOBILE_CALLS *m_calls,
    TIME time
)
{
M_CALL *call;
FRAME *frame;
```

```
    if ((call = get_running_calls ()) == NULL) /* This should never happen */
    {
        root_message ("Error:  Mobile scheduled to send with no calls");
        terminate_sim ();
    }
    if (get_b_call (get_f_call (call)) ≠ call)
    {
        root_message ("MOBILE: Forward link corrupt %x %x %x", call→session,
                (get_f_call (call))→session,
                (get_b_call (get_f_call (call)))→session);
        terminate_sim ();
    }
    if (get_b_call (get_f_call (call)) ≠ call)
    {
        root_message ("Backward link corrupt %x %x %x", call→session,
                (get_b_call (call))→session,
                (get_f_call (get_b_call (call)))→session);
        terminate_sim ();
    }
    if (get_generate_time (call) == time) /* In case current call terminated */
    {
        new_frame (&frame);
        dbg (root_message ("MOBILE: Sending data on %x (%x)", call→session,
                call);)
        ms_data_request (call→session, frame, 0, time);
        set_next_generate (call, CELL_INFO / m_calls_data.sample_rate);
        if (ismaster(call) && get_generate_time (call) > get_terminate (call))
            activate ((EV_FUNC) m_terminate_call,
                    get_terminate (call) < time ? time : get_terminate (call),
                    call);
        set_running_calls (get_f_call (call));
    }
    return (get_generate_time (get_running_calls ()));
}

PUBLIC void ms_data_indication
(
    M_CALL *call,
    FRAME *frame,
    int size,
    TIME time
)
{
    dbg (root_message ("MOBILE: Received data");)
    frame_free (frame);
    return;
}

PUBLIC TIME m_terminate_call
(
    M_CALL *call,
    TIME time
)
```

```
{
    dbg (root_message ("MOBILE: Call completing (1), session %x (%x)",
            call→session, call);)
    set_status (call, get_status (call) | M_CLEAR_STATE);
    ms_release_request (call→session, MS_CLEAR, time);
    unlink_call (call);
    return (NO_EVENT);
}

PUBLIC void ms_release_indication
(
    M_CALL *call,
    int reason,
    TIME time
)
{
    dbg (root_message ("MOBILE: Call completing (2), session %x",
            call→session);)
    unlink_call (call);
    ms_release_response (call→session, MS_CLEAR, time);
    m_call_free (call);
}

PUBLIC void ms_release_confirm
(
    M_CALL *call,
    int reason,
    TIME time
)
{
    dbg (root_message ("MOBILE: Call completing (3) %x", call→session);)
    m_call_free (call);
}

PRIVATE void unlink_call
(
    M_CALL *call
)
{
    if (get_forward (call) ≠ NULL)
        link_backward (get_forward (call), get_backward (call));
    if (get_backward (call) ≠ NULL)
        link_forward (get_backward (call), get_forward (call));
    else
        set_calls_list (get_forward (call));

    /* Unlink the call from the running calls list, if required */
    if (get_f_call (call) ≠ NULL)
    {
        if (get_running_calls () == call)
        {
            if (get_f_call (call) == call)
            {
```

```
                set_running_calls (NULL);
                if (deactivate ((EV_FUNC) send_mobile_cell, &m_calls_data))
                {
                    root_message ("MOBILE: Failed to deactivate only call");
                    terminate_sim ();
                }
            }
            else
            {
                set_running_calls (link_f_call (get_b_call (call),
                        get_f_call (call)));
                link_b_call (get_f_call (call), get_b_call (call));
            }
        }
        else
        {
            link_b_call (get_f_call (call), get_b_call(call));
            link_f_call (get_b_call (call), get_f_call(call));
        }
    }
    link_forward (call, NULL);
    link_backward (call, NULL);
    link_f_call (call, NULL);
    link_b_call (call, NULL);
}

PRIVATE void mobile_call_insert
(
    M_CALL *call,
    TIME time
)
{
M_CALL *w_call;

    init_generate (call, time);
    if ((w_call = get_running_calls ()) == NULL)
    {
        set_running_calls (call);
        link_f_call (link_b_call (call, call), call);
        activate ((EV_FUNC) send_mobile_cell, time, &m_calls_data);
    }
    else
    {
        if (get_generate_time (w_call) > time)
        {
            deactivate ((EV_FUNC) send_mobile_cell, &m_calls_data);
            link_b_call (call, get_b_call (w_call));
            link_f_call (call, w_call);
            link_f_call (get_b_call (call), link_b_call (w_call, call));
            set_running_calls (call);
            activate ((EV_FUNC) send_mobile_cell, time, &m_calls_data);
            dbg (root_message ("rescheduled mobile generator");)
        }
```

```
        else
        {
            while (get_generate_time (get_b_call (w_call)) > time)
                w_call = get_b_call (w_call);
            link_b_call (call, get_b_call (w_call));
            link_f_call (call, w_call);
            link_f_call (get_b_call (call), link_b_call (w_call, call));
        }
    }
    dbg (root_message ("MDQ: %.7f, %.7f, %.7f (%.7f)",
            get_generate_time (get_b_call (call)),
            get_generate_time (call),
            get_generate_time (get_f_call (call)),
            get_generate_time (get_running_calls ()));)
}

/*************** HANDOFF MECHANISMS ******************/
PUBLIC void gen_next_handoff
(
    M_CALL *call,
    TIME time
)
{
TIME when;
RADIO_CHANNEL *where;
int i;

    when = time + distribution (&m_calls_data, handoff);
    i = gen_new_footprint ();
    where = m_calls_data.session_table[i].handoff->rad_chan;
    if (when + where->io_chan.look_ahead > call->time_term)
        return; /* Dont inform of handoff when call will have terminated */
    activate ((EV_FUNC) inform_handoff, when, call);
    call->handoff_on = where;
    call->handoff_at = when + where->io_chan.look_ahead;
    dbg (root_message (
            "MOBILE: Warn handoff time is %.7f, to %x, real %.7f, (%x)",
            when, where, call->handoff_at, call);)
    return;
}

PUBLIC TIME do_handoff_from
(
    M_CALL *call,
    TIME time
)
{
    if (get_terminate (call) < get_generate_time (call))
    {
        deactivate ((EV_FUNC) m_terminate_call, call);
    }
    dbg (root_message ("MOBILE: Call handed off @ %.7f", time);)
    unlink_call (call);
```

```
    /* #### Send call state information #### */

    /* Create a new session connexion */
    ms_handoff_request (call→session, HANDOFF_LEAVING, time, call,
            call→handoffs_done);
    /* The following free message should be after the handoff message
     * protocol has been observed (?)
     */
    m_call_free (call);
    return (NO_EVENT);
}

PUBLIC TIME do_handoff_to
(
    M_CALL *call,
    TIME time
)
{
    call→session = ms_handoff_request (call→session, HANDOFF_ARRIVING, time,
            call, call→handoffs_done);
    while (get_generate_time (call) < time)
        set_next_generate (call, CELL_INFO / m_calls_data.sample_rate);
    dbg (root_message ("MOBILE: Call handed off to here @ %.7f", time);)
    dbg (root_message ("MOBILE: inserting handoff call in run list @ %.7f",
            get_generate_time (call));)
    mobile_call_insert (call, get_generate_time (call));
    gen_next_handoff (call, time);
    return (NO_EVENT);
}

PUBLIC TIME inform_handoff
(
    M_CALL *call,
    TIME time
)
{
char hand_off_imm = HANDOFF_IMM_TYPE;

    dbg (root_message ("Sending warning of handoff at time %.7f %x (%x)", time,
            call→session, call);)
    out_trans (&call→handoff_on→io_chan.io_queue, &hand_off_imm,
            sizeof (hand_off_imm));
    out_trans (&call→handoff_on→io_chan.io_queue, &time, sizeof (TIME));
    out_trans (&call→handoff_on→io_chan.io_queue, &call→handoff_at,
            sizeof (call→handoff_at));
    out_trans (&call→handoff_on→io_chan.io_queue, &call→master,
            sizeof (call→master));
    out_trans (&call→handoff_on→io_chan.io_queue, &call→time_term,
            sizeof (call→time_term));
    out_trans (&call→handoff_on→io_chan.io_queue, &call→session,
            sizeof (call→session));
    out_trans (&call→handoff_on→io_chan.io_queue, &call→time_next,
            sizeof (call→time_next));
```

```
    out_trans (&call→handoff_on→io_chan.io_queue, &call→handoffs_done,
            sizeof (call→handoffs_done));
    trans_flush (&call→handoff_on→io_chan.io_queue);
    call→handoff_on→io_chan.last_time_out = time;
    activate ((EV_FUNC) do_handoff_from, call→handoff_at, call);
    return (NO_EVENT);
}

/* Sched_handoff () returns 1 on failure (there is no memory left)
 * this permits deadlock avoidance to be undertaken.
 */

PUBLIC int sched_handoff
(
    RADIO_CHANNEL *rad_chan,
    NET_PACKET *packet,
    TIME time
)
{
M_CALL *call;
TIME handoff_at;

    if ((call = m_call_alloc ()) == NULL)
        return (1);
    add_calls_list (call);
    in_trans (packet, &handoff_at, sizeof (handoff_at));
    in_trans (packet, &call→master, sizeof (call→master));
    in_trans (packet, &call→time_term, sizeof (call→time_term));
    in_trans (packet, &call→session, sizeof (call→session));
    in_trans (packet, &call→time_next, sizeof (call→time_next));
    in_trans (packet, &call→handoffs_done, sizeof (call→handoffs_done));
    call→handoffs_done++;
    dbg (root_message (
"Scheduling handoff at time %.7f;\n session is %x;\n master?  %c\n term
%.7f\n\
 next %.7f",
            handoff_at, call→session, call→master ? 'Y' : 'N',
            call→time_term, call→time_next);)
    activate ((EV_FUNC) do_handoff_to, handoff_at, call);
    return (0);
}

/************ SESSION LAYER FUNCTIONS ****************/

PRIVATE void show_status
(
    M_SESSION *session,
    TIME time
)
{
unsigned long status;
char stat_str[100];
```

```
    status = get_status (session);
    root_message ("MOBILE: Status of session %x at %.7f:",
            session→my_session_id, time);
    sprintf (stat_str, " STATUS = %x:  ", status);
    if (status & S_HANDOFF_T_STATE)
        strcat (stat_str, "S_HANDOFF_T ");
    if (status & S_HANDOFF_F_STATE)
        strcat (stat_str, "S_HANDOFF_F ");
    if (status & S_HANDOFF_E_STATE)
        strcat (stat_str, "S_HANDOFF_E ");
    if (status & S_CLEAR_STATE)
        strcat (stat_str, "S_CLEAR ");
    if (status & S_RUNNING_STATE)
        strcat (stat_str, "S_RUNNING ");
    if (status & S_SETUP_STATE)
        strcat (stat_str, "S_SETUP ");
    if (status & S_DELETE_STATE)
        strcat (stat_str, "S_DELETE ");
    root_message (stat_str);
    root_message (" Session @ %x", session);
    root_message (" Call @ %x", session→call);
    root_message (" Remote session is %x", session→your_session_id);
    root_message (" Local Handoffs:  %d.  Remote Handoffs %d.",
            session→handoff_no_txed, session→handoff_no_rxed);
    root_message (" %s", session→active ? "Active" : "Passive");
}

/* ms_connect_request ()
 *
 * This proceedure is called by the calling-mobile wishes to create a
 * connexion to another mobile. It creates a local session structure and
 * then sends a MS_CONNECT.REQ frame to the mobile session layer at the
 * remote end.
 */

PUBLIC void ms_connect_request
(
    unsigned long address,
    M_CALL *call,
    TIME term,
    TIME time
)
{
M_SESSION *session;
int your_footprint;
union
{
    FRAME *frame;
    struct m_setup_msg
    {
        int mtype;
        unsigned long my_session_id;
        int your_footprint;
```

```
        unsigned long address;
        TIME term;
    } *data;
} fblock;

    new_session (&session, SES_GENERATE, source_id);
    if (session == NULL) /* No local session available */
    {
        root_message ("MOBILE: Insufficient local sessions");
        ms_connect_confirm (call, MS_FAILED, 0L, time);
        return;
    }
    set_status (session, S_SETUP_STATE);
    your_footprint = locate_mobile (address); /* First guess */
    dbg (root_message ("MOBILE: Call to %d", your_footprint);)
    session→call = call;
    session→active = TRUE; /* I am handling this end of the call */
    new_frame (&fblock.frame);
    fblock.data→mtype = M_SETUP_REQ;
    fblock.data→my_session_id = session→my_session_id;
    fblock.data→your_footprint = your_footprint;
    fblock.data→address = address;
    fblock.data→term = term;
    /* now transmit the cell to the destination and wait for a reply */
    (*mac_f_tx) (fblock.frame, sizeof (struct m_setup_msg) / sizeof (int),
            MOBILE_SETUP_FRAME, m_calls_data.setup_mes_len,
            your_footprint, time);
}

/* ms_connect_response ()
 *
 * This proceedure is called by the called-mobile in response to receiving
 * a MS_CONNECT.REQ frame, and gives the result of this message. If the
 * call is accepted then a MS_FOOTPRINT.ADD message is sent to this session,
 * via the network, to establish the footprint with the correct session
 * numbers. If the call is rejected then the MS_CONNECT.RESP message is
 * returned with the failure reason, and the footprint is not established.
 */

PUBLIC void ms_connect_response
(
    unsigned long session_no,
    M_CALL *call,
    int state,
    TIME time
)
{
M_SESSION *session;
union
{
    FRAME *frame;
    struct m_setup_resp
    {
```

```
        int mtype;
        unsigned long my_session_id;
        unsigned long your_session_id;
        int state;
    } *data;
} fblock;
union
{
    FRAME *frame;
    struct m_footprint_msg
    {
        int state;
        unsigned long my_session_id;
        unsigned long your_session_id;
    } *data;
} fftprnt;

    if ((session = which_session (session_no)) == NULL)
    {
        root_message ("Error during connect response - no session");
        terminate_sim ();
    }
    if (get_status (session) ≠ S_SETUP_STATE)
    {
        root_message (
                "MOBILE: Session (%x) not in setup state when sending
response",
                session→my_session_id);
        show_status (session, time);
        terminate_sim ();
    }
    if (state ≠ MS_FAILED)
    {
        dbg (root_message ("MOBILE: Footprint creation (%x-%x)",
                session→my_session_id, session→your_session_id);)
        new_frame (&fftprnt.frame);
        fftprnt.data→state = M_CREATE_FOOTPRINT;
        fftprnt.data→my_session_id = session→my_session_id;
        fftprnt.data→your_session_id = session→your_session_id;
        (*mac_f_tx) (fftprnt.frame, sizeof (struct m_footprint_msg) /
                sizeof (int), MOBILE_FOOTPRINT, m_calls_data.footprint_len,
                get_footprint (session→my_session_id), time);
        session→call = call;
    }
    new_frame (&fblock.frame);
    fblock.data→mtype = M_SETUP_ACK;
    fblock.data→my_session_id = session→my_session_id;
    fblock.data→your_session_id = session→your_session_id;
    fblock.data→state = state;
    (*mac_f_tx) (fblock.frame, sizeof (struct m_setup_resp) / sizeof (int),
            MOBILE_SETUP_ACK, m_calls_data.setup_ack_len,
            get_footprint (session→your_session_id), time);
    if (state == MS_FAILED)
```

```
            unlink_session (session);
        else
            set_status (session, S_RUNNING_STATE);
    }

    PRIVATE void send_a_m_cell
    (
        int footprint,
        FRAME *frame,
        int size,
        NET_CHANNEL *net_chan,
        TIME time
    )
    {
    CELL *cell;

        if ((cell = cell_alloc ()) == NULL)
        {
            root_message ("MOBILE: Unable to generate cell");
            terminate_sim ();
        }
        cell->birth = time;
        cell->now = time;
        (*make_header) (&cell->header1, &cell->header2, BROADCAST_CELL, footprint);
        cell->body = frame;
        cell->body_size = size;
        if (debug & 0x800)
        {
            add_trace (cell, time, -2);
            cell->trace_size = 1;
        }
        else
        {
            cell->trace_size = 0;
            cell->trace_packet = NULL;
        }

        (*net_chan->o_enq_func) (net_chan, cell, time);
    }

    /* ms_data_request ()
     *
     * Called by either mobile to send a data sample, the call must be in the
     * running state, or an error is generated.
     */

    PUBLIC void ms_data_request
    (
        unsigned long session_no,
        FRAME *frame,
        int size,
        TIME time
    )
```

```
{
M_SESSION *session;
struct ms_data_req
{
    unsigned long your_session_id;
} *data;

    if ((session = which_session (session_no)) == NULL)
    {
        root_message ("Error during data transmission - no session (%x)",
                session_no);
        terminate_sim ();
    }
    if ((get_status (session) & S_RUNNING_STATE) ≠ S_RUNNING_STATE)
    {
        root_message (
                "MOBILE: Attempt to send cell on non-running session (%x)",
                session→my_session_id);
        show_status (session, time);
        terminate_sim ();
    }
    data = (struct ms_data_req *) (((int *) frame) + size);
    data→your_session_id = session→your_session_id;
    dbg (root_message ("Data on %x - %x", session→my_session_id,
            session→your_session_id);)
    send_a_m_cell (get_footprint (session→your_session_id), frame,
            size + sizeof (struct ms_data_req), m_calls_data.channel, time);
}

/* ms_release_request ()
 *
 * Sent by the calling-mobile to terminate a call. To simply the protocol
 * only the calling mobile may actually send this message (the called mobile
 * can send a request for this message to be issued, if required) this
 * prevents possible problems with termination messages crossing and creating
 * odd states within the session protocol.
 * A MS_RELEASE.REQ message is sent to the called mobile requesting that it
 * terminate the call. It is an error for the session not to be in the
 * S_RUNNING_STATE state, although a handoff may be outstanding.
 */

PUBLIC void ms_release_request
(
    unsigned long session_no,
    int reason,
    TIME time
)
{
M_SESSION *session;
union
{
    FRAME *frame;
    struct m_clear_req
```

```
        {
            int mtype;
            unsigned long your_session_id;
            int reason;
        } *data;
    } fblock;

    if ((session = which_session (session_no)) == NULL)
    {
        root_message ("Error during disconnect request - no session");
        terminate_sim ();
    }
    dbg (root_message ("MOBILE: Releasing session %x - %x",
            session→my_session_id, session→your_session_id);)
    if ((get_status (session) & S_RUNNING_STATE) ≠ S_RUNNING_STATE)
    {
        root_message (
                "MOBILE: Release request when session (%x) not in running
state",
                session→my_session_id);
        show_status (session, time);
        terminate_sim ();
    }
    set_status (session, (get_status (session) & ~S_RUNNING_STATE) |
            S_CLEAR_STATE);
    new_frame (&fblock.frame);
    fblock.data→mtype = M_CLEAR_REQ;
    fblock.data→your_session_id = session→your_session_id;
    fblock.data→reason = reason;
    (*mac_f_tx) (fblock.frame, sizeof (struct m_clear_req) / sizeof (int),
            MOBILE_TERM_REQ, m_calls_data.clear_mes_len,
            get_footprint (session→your_session_id), time);
}

/* ms_release_response ()
 *
 * Sent by the called mobile in response to receiving a MS_TERMINATE.REQ
 * message. The session must normally be in the S_CLEAR_STATE state.
 * If the session is still in the S_RUNNING_STATE state then it must also
 * have the S_HANDOFF_T_STATE set, and the termination request was received
 * on the old session: in this case the old session should be in the
 * S_CLEAR_STATE, and have the S_HANDOFF_F_STATE set; since no acknowlegement
 * will ever be received on the new session, both session modules need to be
 * cleared (footprinted sessions for the new session do not exist).
 * If the session is in the S_CLEAR_STATE then an old session may still exist
 * due to the FOOTPRINT.DEL message still being in transit through the
 * network: in this case the session will have the S_HANDOFF_T_STATE cleared,
 * but the S_HANDOFF_F_STATE will still be set for the old session.
 */

PUBLIC void ms_release_response
(
    unsigned long session_no,
```

```
    int status,
    TIME time
)
{
M_SESSION *session;
M_SESSION *old_ses;
union
{
    FRAME *frame;
    struct ms_clear_resp
    {
        int mtype;
        unsigned long your_session_id;
        int status;
    } *data;
} fblock;

    if ((session = which_session (session_no)) == NULL)
    {
        root_message ("Error during release response - no session");
        terminate_sim ();
    }
    dbg (root_message ("MOBILE: Release response %x", session_no);)
    dbg (show_status (session, time);)
    if ((get_status (session) & S_CLEAR_STATE) ≠ S_CLEAR_STATE)
    {
        /* release request must have come on old session */
        if (((old_ses = session→old_session) == NULL) ||
                ((get_status (old_ses) & (S_CLEAR_STATE | S_HANDOFF_F_STATE))
                ≠ (S_CLEAR_STATE | S_HANDOFF_F_STATE)))
        {
            root_message (
          "MOBILE: Not in clear state when sending terminate ack on session
%x",
                session→my_session_id);
            show_status (old_ses, time);
            terminate_sim ();
        }
        /* OK, so delete old session */
        old_ses→new_session = NULL;
        session→old_session = NULL;
        dbg (root_message ("MOBILE: Deleting old session (%x)",
                old_ses→my_session_id);)
        if (get_status (old_ses) ≠ S_DELETE_STATE)
        {
            set_status (old_ses, S_DELETE_STATE);
            activate ((EV_FUNC) release_session, time +
                    m_calls_data.del_session_wait, old_ses);
        }
    }
    if (session→new_session ≠ NULL)
    {
        root_message (
```

```
                "MOBILE: session (%x) has a new session (%x) during clear",
                session→my_session_id, session→new_session→my_session_id);
        terminate_sim ();
    }
    new_frame (&fblock.frame);
    fblock.data→mtype = M_CLEAR_ACK;
    fblock.data→your_session_id = session→your_session_id;
    fblock.data→status = status;
    (*mac_f_tx) (fblock.frame, sizeof (struct ms_clear_resp) / sizeof (int),
            MOBILE_TERM_ACK, m_calls_data.clear_ack_len,
            get_footprint (session→your_session_id), time);
    if (session→old_session ≠ NULL)
    {
        dbg (root_message ("MOBILE: warning - old session (%x) still exists",
                session→old_session→my_session_id);)
        if ((get_status (session→old_session) ≠ S_DELETE_STATE) &&
                (get_status (session→old_session) & S_HANDOFF_F_STATE) ≠
                S_HANDOFF_F_STATE)
        {
            root_message ("MOBILE: Old session (%x) not cleared properly",
                    session→old_session→my_session_id);
            show_status (session→old_session, time);
            terminate_sim ();
        }
        session→old_session→new_session = NULL;
        session→old_session = NULL;
    }
    if (get_status (session) ≠ S_DELETE_STATE)
    {
        set_status (session, S_DELETE_STATE);
        activate ((EV_FUNC) release_session, time +
                m_calls_data.del_session_wait, session);
    }
}

/* mobile_handoff_req ()
 *
 * This proceedure is called by the MAC layer when a MS_HANDOFF.REQ message
 * is received. The session should not be in the S_SETUP_STATE. If the
 * session is not active then the signal is ignored. If the session is in
 * the S_CLEAR_STATE then the signal is ignored and the reply message is not
 * sent, otherwise the response message is sent. If there is a new session
 * then the reply is sent using the new session id.
 */

PUBLIC void mobile_handoff_req
(
    FRAME *frame,
    int size,
    TIME time
)
{
M_SESSION *session;
```

```
union
{
    FRAME *frame;
    struct ms_handoff
    {
        int mtype;
        unsigned long my_old_session_id;
        unsigned long my_new_session_id;
        unsigned long your_session_id;
        int my_ho_cnt;
    } *data;
} fblock, fresp;

    fblock.frame = frame;
    if (!in_footprint (get_footprint (fblock.data→your_session_id)))
    {
        frame_free (frame);
        return; /* Not really in this footprint */
    }
    if ((session = which_session (fblock.data→your_session_id)) == NULL)
    {
        /* Under VERY rare circumstances (handoffs occuring at both ends) it
         * is just possible for a HANDOFF.REQ message to arrive on a new
         * session before the HANDOFF.RESP message has returned - since the
         * two messages come from different places. This only affects the
         * periphery of the footprint where the new sessions have yet to be
         * created, but the HANDOFF.REQ message contains enough information
         * to build a new session - and we must check for the HANDOFF.RESP
         * message arriving later. The temporary session is never active
         * so no message will be sent to the other side.
         */

        root_message (
                "*********************************************************");
        root_message ("* MOBILE: Session (%6x) doesn't exist during handoff *",
                fblock.data→your_session_id);
        root_message (
                "* Creating temporary session *");
        root_message (
                "*********************************************************");
        new_session (&session, SES_FORCE, fblock.data→your_session_id);
        if (session == NULL)
        {
            root_message ("MOBILE: Unable to create_session");
            terminate_sim ();
        }
        set_status (session, S_RUNNING_STATE | S_HANDOFF_E_STATE);
        session→handoff_no_txed = 0; /* Don't know this */
        session→handoff_no_rxed = fblock.data→my_ho_cnt;

    }
    dbg (root_message ("MOBILE: received handoff request @ %.7f", time);)
    if (get_status (session) == S_DELETE_STATE)
```

```
        root_message ("MOBILE: Rxed handoff on session (%x) in delete state",
                session->my_session_id);
    if (session->handoff_no_rxed > fblock.data->my_ho_cnt)
    {
        dbg (root_message (
                "MOBILE: Bogus count number in req -ignored request");
                show_status (session, time);)
        frame_free (frame);
        return;
    }
    session->your_session_id = fblock.data->my_new_session_id;
    session->handoff_no_rxed = fblock.data->my_ho_cnt;
    if (session->new_session) /* Have I done a handoff? */
    {
        dbg (root_message (
                "MOBILE: Outstanding handoff during remote handoff");)
        session = session->new_session;
        session->your_session_id = fblock.data->my_new_session_id;
        session->handoff_no_rxed = fblock.data->my_ho_cnt;
    }
    if ((get_status (session) & S_CLEAR_STATE) == S_CLEAR_STATE)
    {
        dbg (root_message (
                "MOBILE: Session is clearing when handoff received");)
        frame_free (frame);
        return; /* Don't send reply, we are terminating */
    }
    if (session->active)
    {
        /* Correct the entry for my session id, in case I have moved */
        fblock.data->your_session_id = session->my_session_id;
        fblock.data->my_ho_cnt = session->handoff_no_txed;
        (*mac_f_tx) (fblock.frame, sizeof (struct ms_handoff) / sizeof (int),
                MOBILE_HANDOFF_RESP, m_calls_data.handoff_len,
                get_footprint (session->your_session_id), time);
    }
    else
        frame_free (frame);
    dbg (root_message ("MOBILE: My session:  %x, His old %x His new %x",
            session->my_session_id, fblock.data->my_old_session_id,
            fblock.data->my_new_session_id);)
}

/* mobile_handoff_resp ()
 *
 * This procedure is called by the MAC layer when a MS_HANDOFF.CONF message
 * is received. If the session exists (ie this node initiated the
 * handoff) then it must be in either the S_RUNNING_STATE state or the
 * S_CLEAR_STATE state; in both cases the S_HANDOFF_T_STATE flag must be set:
 * the handoff state is cleared and a FOOTPRINT.DEL message is sent to the
 * old footprint to delete it. If the session does not exist then a new one
 * is created with that number and set to the S_RUNNING_STATE state.
 */
```

```
PUBLIC void mobile_handoff_resp
(
    FRAME *frame,
    int size,
    TIME time
)
{
M_SESSION *session;
union
{
    FRAME *frame;
    struct ms_handoff
    {
        int mtype;
        unsigned long my_old_session_id;
        unsigned long my_new_session_id;
        unsigned long your_session_id;
        int my_ho_cnt;
    } *data;
} fblock;
union
{
    FRAME *frame;
    struct m_footprint_msg
    {
        int state;
        unsigned long my_session_id;
        unsigned long your_session_id;
    } *data;
} fftprnt;

    fblock.frame = frame;
    if (!in_footprint (get_footprint (fblock.data→my_new_session_id)))
    {
        frame_free (frame);
        return; /* Not really in this footprint */
    }
    if ((session = which_session (fblock.data→my_new_session_id)) == NULL)
    {
        dbg (root_message (
"MOBILE: New session %x (old %x) in response to handoff @ %.7f",
                fblock.data→my_new_session_id,
                fblock.data→my_old_session_id, time);)
        new_session (&session, SES_FORCE, fblock.data→my_new_session_id);
        if (session == NULL)
        {
            root_message ("Unable to create session");
            terminate_sim ();
        }
        set_status (session, S_RUNNING_STATE);
        session→your_session_id = fblock.data→your_session_id;
        session→handoff_no_rxed = fblock.data→my_ho_cnt;
```

```
        session→handoff_no_txed = 0; /* Don't know this yet */
    }
    else /* We are running this session, so make sure it is right */
    {
        dbg (root_message ("MOBILE: received handoff response @%.7f %x %x (%d)",
                time, session→my_session_id, fblock.data→your_session_id,
                fblock.data→my_ho_cnt);)
        if (session→my_session_id ≠ fblock.data→my_new_session_id)
        {
            root_message ("MOBILE: error!  sessions do not match");
            terminate_sim ();
        }
        if (session→handoff_no_rxed > fblock.data→my_ho_cnt)
        {
            if ((get_status (session) & S_HANDOFF_E_STATE) ≠ S_HANDOFF_E_STATE)
            {
                if ((get_status (session) & S_HANDOFF_T_STATE) ==
                        S_HANDOFF_T_STATE)
                {
                    dbg (root_message ("MOBILE: Using that 'goto'!!!!");)
                    goto MAJOR_KLUDGE_FIX; /******* YUK! ********/
                }
                dbg (root_message (
                        "MOBILE: Bogus count number in ack -ignored request");
                        show_status (session, time);)
            }
            else
            {
                /* We have recovered from the special case that occured in
                 * mobile_handoff_req (). NOTE: We must NOT update
                 * "your_session_id" because the message contains an out-of-
                 * date copy.
                 */
                root_message (
                        "***********************************************");
                root_message ("* MOBILE: Session (%6x) has been recovered *",
                        session→my_session_id);
                root_message (
                        "***********************************************");
                set_status (session, S_RUNNING_STATE);
            }
        }
        else
        {
            if (((get_status (session) & S_SETUP_STATE) ≠ 0) ||
                    ((get_status (session) & S_HANDOFF_T_STATE) ≠
                    S_HANDOFF_T_STATE))
            {
                if (session→handoff_no_rxed < fblock.data→my_ho_cnt)
                {
                    dbg (root_message (
                  "MOBILE: Multiple handoff acks - updating 'your_session_id'");
                            show_status (session, time);)
```

```
                    session→handoff_no_rxed = fblock.data→my_ho_cnt;
                    session→your_session_id = fblock.data→your_session_id;
                    frame_free (frame);
                    return;
                }
                dbg (root_message (
                    "MOBILE: Multiple handoff acks - ignoring old/same aged");
                    show_status (session, time);)
                frame_free (frame);
                return;
            }
            session→handoff_no_rxed = fblock.data→my_ho_cnt;
            if (session→your_session_id ≠ fblock.data→your_session_id)
            {
                dbg (show_status (session, time);
                    root_message (
"MOBILE: (%x) remote session ids do not match - replacing %x (%d) with %x
(%d)",
                    session→my_session_id, session→your_session_id,
                    session→handoff_no_rxed, fblock.data→your_session_id,
                    fblock.data→my_ho_cnt);)
            }
            session→your_session_id = fblock.data→your_session_id;
MAJOR_KLUDGE_FIX:
            if (session→active) /* Only sent by the active session */
            {
                set_status (session, get_status (session) & ~S_HANDOFF_T_STATE);
                new_frame (&fftprnt.frame);
                fftprnt.data→state = M_DELETE_FOOTPRINT;
                fftprnt.data→my_session_id = fblock.data→my_old_session_id;
                fftprnt.data→your_session_id = session→your_session_id;
                (*mac_f_tx) (fftprnt.frame, sizeof (struct m_footprint_msg) /
                        sizeof (int), MOBILE_FOOTPRINT,
                        m_calls_data.footprint_len,
                        get_footprint (fblock.data→my_old_session_id), time);
            }
        }
    }
    frame_free (frame);
}

/* ms_handoff_request ()
 *
 * Called by the mobile to initiate a handoff. One of two states may be
 * specified: if the reason is that the call is leaving then the session is
 * passivated and the S_HANDOFF_F_STATE state added (the call must be in the
 * running state with no outstanding local handoff); otherwise the call implies
 * that a mobile is moving to this node and that a new session is required,
 * a new session is created in the S_RUNNING_STATE with the S_HANDOFF_T_STATE
 * attribute and a MS_HANDOFF.REQ message sent to the remote mobile. The old
 * session is linked to the new one and the S_HANDOFF_F_STATE attribute is
 * added.
 */
```

```
PUBLIC unsigned long ms_handoff_request
(
    unsigned long session_no,
    int reason,
    TIME time,
    M_CALL *call,
    int handoffs_done
)
{
M_SESSION *session;
M_SESSION *new_ses;
union
{
    FRAME *frame;
    struct ms_handoff
    {
        int mtype;
        unsigned long my_old_session_id;
        unsigned long my_new_session_id;
        unsigned long your_session_id;
        int my_ho_cnt;
    } *data;
} fblock;

    if ((session = which_session (session_no)) == NULL)
    {
        root_message ("Error during handoff request - no session");
        terminate_sim ();
    }
    switch (reason)
    {
    case HANDOFF_LEAVING:
        if (get_status (session) ≠ S_RUNNING_STATE)
        {
            root_message ("MOBILE: bad status during handoff request (%x)",
                    session);
            show_status (session, time);
            terminate_sim ();
        }
/* session->active = FALSE; */
        session→call = NULL;
        set_status (session, get_status (session) | S_HANDOFF_F_STATE);
        return (session→my_session_id);
    case HANDOFF_ARRIVING:
        if (get_status (session) ≠ S_RUNNING_STATE)
        {
            root_message (
                    "MOBILE: session is not running with handoff request (%x)",
                    session→my_session_id);
            show_status (session, time);
            terminate_sim ();
        }
```

```
            new_session (&new_ses, SES_GENERATE, source_id);
            if (new_ses ≠ NULL)
            {
                set_status (new_ses, S_RUNNING_STATE | S_HANDOFF_T_STATE);
                set_status (session, get_status (session) | S_HANDOFF_F_STATE);
                new_frame (&fblock.frame);
                fblock.data→mtype = M_HANDOFF_REQ;
                fblock.data→my_old_session_id = session→my_session_id;
                fblock.data→my_new_session_id = new_ses→my_session_id;
                fblock.data→your_session_id = session→your_session_id;
                fblock.data→my_ho_cnt = new_ses→handoff_no_txed = handoffs_done;
                new_ses→your_session_id = session→your_session_id;
                new_ses→old_session = session;
                session→new_session = new_ses;
                session→call = new_ses→call = call;
                new_ses→active = session→active = TRUE;
                dbg (root_message ("MOBILE: Sending handoff cmd %x, %x, %x @ %.7f",
                        new_ses→my_session_id, session→my_session_id,
                        new_ses→your_session_id, time);)
                (*mac_f_tx) (fblock.frame, sizeof (struct ms_handoff) / sizeof(int),
                        MOBILE_HANDOFF_REQ, m_calls_data.handoff_len,
                        get_footprint (new_ses→your_session_id), time);

                /* The old session is not deleted until the reply is received,
                 * since data may still be arriving on the old session identifier.
                 *
                 * However, new data is sent using the new session id.
                 */
                return (new_ses→my_session_id);
            }
            else
            {
                root_message ("MOBILE: No free sessions during handoff");
                terminate_sim ();
            }
        default:
            root_message ("MOBILE: Bad handoff type request");
            terminate_sim ();
        }
}

/* mobile_setup_req ()
 *
 * Called by the MAC layer on receipt of a MS_HANDOFF.REQ message. If the
 * mobile is connected to this source (mobile_in_footprint) then the
 * routine tries to allocate a session, otherwise it does nothing; if
 * sucessful passes the remainder of the message upstairs (ie to the mobile
 * part) otherwise a reject message is sent. The call is put into the
 * S_SETUP_STATE.
 */

PUBLIC void mobile_setup_req
(
```

```
    FRAME *frame,
    int size,
    TIME time
)
{
M_SESSION *session;
union
{
    FRAME *frame;
    struct m_setup_msg
    {
        int mtype;
        unsigned long my_session_id;
        int your_footprint;
        unsigned long address;
        TIME term;
    } *data;
} fblock;
union
{
    FRAME *frame;
    struct m_setup_resp
    {
        int mtype;
        unsigned long my_session_id;
        unsigned long your_session_id;
        int state;
    } *data;
} freply;

    fblock.frame = frame;
    if (!in_footprint (fblock.data→your_footprint))
    {
        frame_free (frame); /* Not really in this footprint */
        return;
    }
    dbg (root_message ("Received connect request (???%04x-%x)",
            fblock.data→your_footprint, fblock.data→my_session_id);)
    if (mobile_in_footprint (fblock.data→your_footprint, fblock.data→address))
    {
        new_session (&session, SES_GENERATE, fblock.data→your_footprint);
        if (session == NULL) /* No sessions left */
        {
            /* We must only send one reject frame => pick the controller */
            new_frame (&freply.frame);
            freply.data→mtype = M_SETUP_ACK;
            freply.data→my_session_id = 0;
            freply.data→your_session_id = fblock.data→my_session_id;
            freply.data→state = MS_FAILED;
            (*mac_f_tx) (freply.frame,
                    sizeof (struct m_setup_resp) / sizeof (int),
                    MOBILE_SETUP_ACK, m_calls_data.setup_ack_len,
                    get_footprint (fblock.data→my_session_id), time);
```

```
            }
            else
            {
                set_status (session, S_SETUP_STATE);
                session→active = TRUE;
                session→your_session_id = fblock.data→my_session_id;
                ms_connect_indication (session→my_session_id, fblock.data→term,
                        time);
            }
        }
        else
        {
            dbg (root_message ("MOBILE: Setup req ignored in footprint");)
        }
        frame_free (fblock.frame);
}

/* mobile_setup_ack ()
 *
 * Called by the MAC layer upon receipt of a MS_CONNECT.CONF message.
 * If the session exists then it must be in the S_SETUP_STATE state.
 * If the message is a rejection then the structure is cleared and the
 * mobile informed. Otherwise a new session is created with the supplied
 * session numbers. All the sessions are put into the S_RUNNING_STATE.
 */

PUBLIC void mobile_setup_ack
(
    FRAME *frame,
    int size,
    TIME time
)
{
M_SESSION *session;
union
{
    FRAME *frame;
    struct m_setup_resp
    {
        int mtype;
        unsigned long my_session_id;
        unsigned long your_session_id;
        int state;
    } *data;
} fblock;

    fblock.frame = frame;
    if (!in_footprint (get_footprint(fblock.data→your_session_id)))
    {
        frame_free (frame); /* Not really in this footprint */
        return;
    }
```

```
        dbg (root_message ("MOBILE: Received setup acknowledge (%x-%x)",
                fblock.data→your_session_id, fblock.data→my_session_id);)
        if ((session = which_session (fblock.data→your_session_id)) == NULL)
        {
            /* In footprint, but not controlling the call */
            if (fblock.data→state == MS_ACCEPT)
            {
                new_session (&session, SES_FORCE, fblock.data→your_session_id);
                set_status (session, S_RUNNING_STATE);
                session→active = FALSE;
                session→my_session_id = fblock.data→your_session_id;
                session→your_session_id = fblock.data→my_session_id;
                session→call = NULL;
            }
        }
        else
        {
            if (get_status (session) ≠ S_SETUP_STATE)
            {
                root_message ("MOBILE: session (%x) not in setup on rx of ACK",
                        session→my_session_id);
                show_status (session, time);
                terminate_sim ();
            }
            if (session→active ≠ TRUE)
            {
                root_message (
                        "Error - previous session (%x-%x) not properly cleared",
                        session→my_session_id, session→your_session_id);
                terminate_sim ();
            }
            session→your_session_id = fblock.data→my_session_id;
            ms_connect_confirm (session→call, fblock.data→state,
                    session→my_session_id, time);
            if (fblock.data→state == MS_FAILED)
                unlink_session (session);
            else
                set_status (session, S_RUNNING_STATE);
        }
        frame_free (fblock.frame);
}

/* mobile_term_req ()
 *
 * Called by the MAC layer upon receipt of a MS_RELEASE.REQ message. The
 * call must be in the S_RUNNING_STATE state, but there may be an outstanding
 * handoff. All the difficult cases are now handled by the acknowledgement
 * message.
 */

PUBLIC void mobile_term_req
(
    FRAME *frame,
```

```
    int size,
    TIME time
)
{
M_SESSION *session;
union
{
    FRAME *frame;
    struct m_term_req
    {
        int mtype;
        unsigned long your_session_id;
        int reason;
    } *data;
} fblock;

    fblock.frame = frame;
    if (!in_footprint (get_footprint(fblock.data->your_session_id)))
    {
        frame_free (frame); /* Not really in this footprint */
        return;
    }
    if ((session = which_session (fblock.data->your_session_id)) == NULL)
    {
        root_message ("Error during terminate indication - no session (%x)",
                fblock.data->your_session_id);
        terminate_sim ();
    }
    dbg (root_message ("MOBILE: mobile term req %x", session->my_session_id);)
    dbg (show_status (session, time);)
    if ((get_status (session) & S_RUNNING_STATE) ≠ S_RUNNING_STATE)
    {
        root_message ("MOBILE: session (%x) not running during terminate req",
                session->my_session_id);
        show_status (session, time);
        terminate_sim ();
    }
    if (session->call) /* If we really are controlling the call */
    {
        set_status (session, (get_status(session) & ~S_RUNNING_STATE) |
                S_CLEAR_STATE);
        dbg (show_status (session, time);)
        ms_release_indication (session->call, fblock.data->reason, time);
        /* Do not unlink the session here - the response message hasn't
         * been sent yet so it will do it
         */
    }
    else
    {
        if (get_status (session) ≠ S_DELETE_STATE)
        {
            set_status (session, S_DELETE_STATE);
            activate ((EV_FUNC) release_session, time +
```

```
                    m_calls_data.del_session_wait, session);
            }
        }
        frame_free (fblock.frame);
    }

    /* mobile_term_ack ()
     *
     * Called by the MAC layer upon receipt of a MS_RELEASE.CONF message.
     * The active node should be in the S_CLEAR_STATE state, otherwise the
     * call must be in the S_RUNNING_STATE; there should not be any
     * handoff outstanding (although the old footprint may still exist if
     * the FOOTPRINT.DEL message is still in transit through the network).
     * The session is cleared and if the call was active the confirmation is
     * passed up to the mobile.
     */

    PUBLIC void mobile_term_ack
    (
        FRAME *frame,
        int size,
        TIME time
    )
    {
    M_SESSION *session;
    union
    {
        FRAME *frame;
        struct m_term_req
        {
            int mtype;
            unsigned long your_session_id;
            int state;
        } *data;
    } fblock;

        fblock.frame = frame;
        if (!in_footprint (get_footprint(fblock.data→your_session_id)))
        {
            frame_free (frame); /* Not really in this footprint */
            return;
        }
        if ((session = which_session (fblock.data→your_session_id)) == NULL)
        {
            root_message ("Error during terminate confirm - no session (%x)",
                    fblock.data→your_session_id);
            terminate_sim ();
        }
        dbg (root_message ("Terminating session (%x-%x)", session→my_session_id,
                session→your_session_id);)
        if (session→active)
        {
            if (get_status (session) ≠ S_CLEAR_STATE)
```

```
        {
            root_message (
                    "MOBILE: active session (%x) not in clear state when acked",
                    session→my_session_id);
            show_status (session, time);
            terminate_sim ();
        }
        ms_release_confirm (session→call, fblock.data→state, time);
    }
    else if (get_status (session) ≠ S_RUNNING_STATE)
    {
        root_message ("MOBILE: Footprint of session (%x) not running at clear",
                session→my_session_id);
        show_status (session, time);
        terminate_sim ();
    }
    /* It is just possible that a handoff has recently occurred, but that
     * the FOOTPRINT.DEL message has not got through the network yet
     * so clean up the connexion to any old sessions
     */
    if (session→old_session)
    {
        session→old_session→new_session = NULL;
        session→old_session = NULL;
    }
    mobile_handoffs += session→handoff_no_txed + session→handoff_no_rxed;
    if (get_status (session) ≠ S_DELETE_STATE)
    {
        set_status (session, S_DELETE_STATE);
        activate ((EV_FUNC) release_session, time +
                m_calls_data.del_session_wait, session);
    }
    frame_free (fblock.frame);
}

PUBLIC TIME release_session
(
    M_SESSION *session,
    TIME time
)
{
    dbg (root_message ("MOBILE: Releasing session %x @ %.7f", session, time);)
    if (get_status (session) ≠ S_DELETE_STATE)
    {
        root_message ("MOBILE: trying to release an active session (%x)",
                session→my_session_id);
        show_status (session, time);
        terminate_sim ();
    }
    if (session→new_session ≠ NULL)
    {
        /* Now release the old session. It will not be used any more */
        session→new_session→old_session = NULL;
```

```
        session→new_session = NULL;
    }
    unlink_session (session);
    return (NO_EVENT);
}

/* mobile_footprint ()
 *
 * Called by the MAC layer on receipt of a FOOTPRINT.? message. ? may be
 * either ADD or DEL. If it is ADD then a new session is created in the
 * run state provided that it does not already exist (if it does then it
 * must be in the run state). If it is DEL then the session (which should
 * be running) is deleted.
 */

PUBLIC void mobile_footprint
(
    FRAME *frame,
    int size,
    TIME time
)
{
M_SESSION *session;
union
{
    FRAME *frame;
    struct m_footprint_msg
    {
        int state;
        unsigned long my_session_id;
        unsigned long your_session_id;
    } *data;
} fblock;

    fblock.frame = frame;
    if (!in_footprint (get_footprint(fblock.data→my_session_id)))
    {
        frame_free (frame); /* Not really in this footprint */
        return;
    }
    dbg (root_message ("MOBILE: Received footprint cmd %d, (%x-%x)",
            fblock.data→state, fblock.data→my_session_id,
            fblock.data→your_session_id);)
    if ((session = which_session (fblock.data→my_session_id)) == NULL)
    {
        dbg (root_message ("New session");)
        if (fblock.data→state == M_CREATE_FOOTPRINT)
        {
            new_session (&session, SES_FORCE, fblock.data→my_session_id);
            if (session == NULL)
            {
                root_message ("Unable to build a footprint session");
                terminate_sim ();
```

```
            }
            set_status (session, S_RUNNING_STATE);
            session→your_session_id = fblock.data→your_session_id;
        }
        else if (fblock.data→state == M_DELETE_FOOTPRINT)
        {
            root_message ("Missing session during footprint delete (%x)",
                    fblock.data→my_session_id);
            terminate_sim ();
        }
    }
    else /* We already have a copy of this session; make sure it is right */
    {
        dbg (root_message ("Old session");)
        if (session→my_session_id ≠ fblock.data→my_session_id)
        {
            root_message ("MOBILE: Error sessions do not match (%x-%x), (%x-%x)",
                    session→my_session_id, session→your_session_id,
                    fblock.data→my_session_id, fblock.data→your_session_id);
            terminate_sim ();
        }
        if (fblock.data→state == M_DELETE_FOOTPRINT)
        {
            if (((get_status (session) & S_RUNNING_STATE) ≠ S_RUNNING_STATE)
                    && (get_status (session) ≠ S_DELETE_STATE))
            {
                root_message (
                        "MOBILE: delete footprint:  session (%x) not running",
                        session→my_session_id);
                show_status (session, time);
                terminate_sim ();
            }
            if (get_status (session) ≠ S_DELETE_STATE)
            {
                set_status (session, S_DELETE_STATE);
                activate ((EV_FUNC) release_session, time +
                        m_calls_data.del_session_wait, session);
            }
        }
        else if ((get_status (session) & S_RUNNING_STATE) ≠ S_RUNNING_STATE)
        {
            root_message ("MOBILE: add footprint:  Old session (%x) not running",
                    session→my_session_id);
            show_status (session, time);
            terminate_sim ();
        }
    }
    frame_free (frame);
}

PUBLIC void mobile_cell_rx
(
    CELL *cell,
```

```
    TIME time
)
{
M_SESSION *session;
struct ms_data_req
{
    unsigned long your_session_id;
} *data;
int size;

    size = cell→body_size - sizeof (struct ms_data_req);
    data = (struct ms_data_req *) (((char *) cell→body) + size);
    if (!in_footprint (get_footprint(data→your_session_id)))
    {
        cell_free (cell); /* Not really in this footprint */
        return;
    }
    if ((session = which_session (data→your_session_id)) == NULL)
    {
        dbg (root_message ("Warning:  data indication - no session (%x)",
                data→your_session_id);)
        cell_free (cell);
        return;
    }
    if (session→call ≠ NULL)
    {
        d_entry (m_calls_data.d_stats, time - cell→birth);
        ms_data_indication (session→call, cell→body, size, time);
    }
    else
    {
        /* Keep a copy of the last data frame received */
        if (session→last_frame)
            frame_free (session→last_frame);
        session→last_frame = cell→body;
        session→last_frame_size = size;
    }
    cell→body_size = 0;
    cell→body = NULL;
    cell_free (cell);
}

PRIVATE void new_session
(
    M_SESSION **session,
    int generate,
    unsigned long ses_id
)
{
int i;
M_SESSION **ses;

    if ((ses = m_calls_data.session_table[get_footprint(ses_id)].sessions) ==
```

```
            NULL)
    {
        root_message ("Request to create session outside footprint (%d)",
                ses_id);
        *session = NULL;
        return;
    }
    if (generate)
    {
        /* Find an unused session */
        for (i = m_calls_data.max_sessions - 1; i ≥ 0 && ses[i] ≠ NULL; i--)
            ;
        if (i < 0)
        {
            *session = NULL;
            return;
        }
    }
    else
    {
        i = get_session_no (ses_id);
        if (ses[i] ≠ NULL)
        {
            root_message ("Trying to duplicate existing session");
            terminate_sim ();
        }
    }
    if ((ses[i] = m_session_alloc ()) == NULL)
    {
        root_message ("Unable to allocate session");
        terminate_sim ();
    }
    *session = ses[i];
    (*session)→call = NULL;
    (*session)→last_frame = NULL;
    (*session)→last_frame_size = 0;
    set_session_id (&(*session)→my_session_id, get_footprint(ses_id), i);
    (*session)→your_session_id = 0;
    (*session)→new_session = (*session)→old_session = NULL;
    (*session)→active = FALSE;
    (*session)→handoff_no_txed = 0;
    (*session)→handoff_no_rxed = 0;
}

PRIVATE void unlink_session
(
    M_SESSION *session
)
{
    dbg (root_message ("MOBILE: Deleting session %x (%x)",
            session→my_session_id, session);)
    m_calls_data.session_table[get_footprint(session→my_session_id)].
            sessions[get_session_no(session→my_session_id)] = NULL;
```

```
    if (session→new_session || session→old_session)
    {
        root_message ("MOBILE: Unlinking partly completed handoff:  %x %x %x",
                session→my_session_id,
                session→new_session ? session→new_session→my_session_id : 0,
                session→old_session ? session→old_session→my_session_id : 0);
        terminate_sim ();
    }
    session→active = FALSE;
    if (session→last_frame)
    {
        frame_free (session→last_frame);
        session→last_frame = NULL;
    }
    m_session_free (session);
}
```