Performance Analysis of Channel Allocation Schemes for Half and Full Rate Calls in GSM

Milosh V. Ivanovich B.E. (Hons)
Department of Computer Science
Monash University

Thesis submitted for examination
for the degree of Master of Computing

October 1995

ACKNOWLEDGEMENTS

I thank my supervisor Dr. Moshe Zukerman for his guidance throughout this research work, particularly with regards to formulation of the overall problem, and for reviewing this document and providing useful comments.

I would also like to thank Prof. Maxim Gitlits and Paul Fitzpatrick for their information about GSM standards development and intracell handover, and Dr. Bob Warfield for his idea on the evaluation of scheme efficiency.

I also greatly thank my father, Dr. Vladimir D. Ivanovic, for proofreading this thesis, and the examiners for their helpful comments.

DECLARATION

This thesis contains no material which has been accepted for the award of any other degree or diploma in any other university, and to the best of my knowledge contains no material previously published or written by another person except where reference has been made in the text of the thesis.

Signed:

Milosh Vladimir Ivanovich
Department of Computer Science
Monash University

ABSTRACT

Manufacturers are following recent standards developments and are currently developing half rate voice coding/decoding equipment (*codecs*) of acceptable quality in order to significantly increase the efficiency of GSM networks. The focus of this research is on the transition period during which some of the users will use the new half rate handsets, while others will hold on to their older full rate handsets, thereby creating a system with a varying mix of two types of traffic: full and half rate users.

This thesis proposes a model for GSM resource management and considers nine channel allocation schemes, with these two types of traffic loading. The nine schemes are: Random, First Fit, Best Fit, Repacking, Repacking with Perpetual Reservation (RPR), Repacking with Perpetual Half Slot Reservation (RPHSR), Repacking with Random Reservation (RRR), Fixed Boundary and Sliding Boundary. The performance of each scheme is determined, based upon the criteria of efficiency, fairness, as well as ease of implementation. Analytic numerical methods are used to investigate each scheme's efficiency and blocking probability behaviour, and this is successfully compared with simulation results. The analytic solution is based on a reduction of the state space to a manageable size using a mapping approach from an *m* dimensional state space down to a two or three dimensional state space.

Initially a preliminary study of blocking probability behaviour and efficiency is carried out for each scheme. The framing structure adopted for this purpose is an eight timeslot frame, without regard to multiple carrier frequencies or any reserved broadcast channels. This study immediately eliminates the Fixed and Sliding Boundary schemes from further consideration due to their extreme unfairness and inefficiency.

The idea behind the eight timeslot frame is then extended to a more realistic model of GSM, by modelling n=2 and n=3 carrier frequency systems, made up of a reserved control/signalling timeslot and $8n$-1 user slots. At the expense of significantly increased CPU resources (time, memory) it was still feasible to analytically evaluate the blocking probability and efficiency of six of the nine schemes, (with the exceptions being First Fit and the two already eliminated Fixed and Sliding Boundary schemes). Comparisons were then made between the original eight timeslot frame model and the more realistic model, where the systems had one, two and three carrier frequencies (with 7, 15 and 23 timeslots respectively). The results show that for each of the six schemes considered, systems of all three sizes perform increasingly better with a higher proportion of half rate customers. It was also found that the higher the

proportion of half rate callers is, the larger is the capacity benefit gained by employing more complex schemes. However, the capacity benefit (i.e. increase) gained by employing the more efficient Repacking family of schemes is generally found to reduce with increasing system size.

An overall comparison of the schemes, taking into account all traffic mixes, is performed with special weight of importance paid to the system with three carrier frequencies (as real network applications are likely to operate on multi-carrier schemes). The comparison is done by way of awarding scores for the three above mentioned criteria and obtaining a total.

The scheme with the highest overall total score is that of Repacking with Perpetual Half Slot Reservation (RPHSR) as it achieves completely equal blocking probabilities for half and full rate users over a wide range of traffic mixes (total fairness), and it achieves the best maximum customer capacity subject to a Grade of Service (GOS) constraint (best efficiency). However, this scheme is not very simple to implement, and because of this, is challenged by the Best Fit scheme, which runs a very close second in the overall score. The Best Fit scheme is significantly simpler to implement, with only slightly worse efficiency than RPHSR, and its only downfall is that it does not assure equal blocking probability for both user types.

These two schemes are the ones most worthy of considering for possible implementation in a real network, and therefore the advantages each scheme has over the other are presented below:

Best Fit

- Simpler to implement.
- No intracell handover affecting voice quality.

RPHSR

- More efficient in utilising network resources.
- Completely fair in terms of blocking probabilities.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

With the current observed growth of demand for digital mobile telephony, increasing the capacity of existing networks is a major priority of the large telecommunications companies (*telcos*). It is clear that increased capacity will allow accommodating more and more new customers, hence bringing in steadily larger revenues. One way of increasing capacity is to install more base stations, which is based on the principle of continued use of current equipment, but in greater numbers. An alternative approach would be to enhance the current equipment so that capacity growth is achieved through better efficiency rather than increased quantity.

The purpose of this thesis is to look at this latter method of increasing network capacity, which may well be more cost effective to implement than pure *new resource allocation*. This method relies on the improvement of existing voice coding technology enabling acceptable quality speech to be encoded into only half the number of bits (per second) which are currently required.

A typical Time Division Multiplexing Scheme (TDMA) can then immediately fit two users into the duration of a timeslot previously allocated to just a single user. In particular, this thesis looks at the Groupe Special Mobile (GSM) standard. GSM later became Global System for Mobile Communications, and is described in detail in the standard specifications [ETSI92], and by Mouly and Pautet [MOUL92].

GSM, unlike a few other systems, is truly global, and has been adopted by many countries (a total of 26 European and 26 non-European countries, including Australia) [PADG95]. It is a system which uses a combination of TDMA and Frequency Division Multiple Access (FDMA) with eight timeslots per radio channel.

The implications of being able to fit two users into a timeslot previously long enough for only one raise two very significant issues which are the primary research objectives of this thesis. Firstly, the telcos and manufacturers of handsets must decide about the way in which this extra capacity in the TDMA frames will be used - how will the calls be *packed.* And secondly, regardless of the rate of customers' take up of any new equipment on offer (such as the new handsets outfitted with *half rate* coders/decoders, *codecs*) there will be existing full rate traffic competing with the 50% leaner half rate traffic - from a teletraffic engineering point of view, do the achievable capacity increases really approach a doubling, as intuition would suggest? Before these questions can be answered, a mathematical model of the GSM framing structure must be defined.

This model then allows solving for the system states under various parametric conditions. Efficiency and blocking probability measures can then be obtained, by suitably varying these input parameters.

The contribution of this thesis is therefore threefold. Firstly, it provides telcos and manufacturers with an extensive analysis of a comprehensive set of available channel allocation schemes, while also giving rise to some novel modifications to known Repacking schemes, [ZUKE88, ZUKE89]. Secondly, guidelines about how to compare the relative merits of each scheme are proposed. And finally, it proposes a state reduction technique, of the type of [KAUF81], [ROBE81] and [ZUKE89], which makes it possible to solve exactly otherwise numerically intractable computational problems.

2. A MODEL OF FULL AND HALF RATE USAGE

2.1 GSM Overview and Standards Development

Before we define the model describing a GSM hybrid full/half rate system, Subsections 2.1.1 and 2.1.2 present the details of those GSM standards which are relevant to, or feature in the model defined in Subsection 2.2.

2.1.1 GSM Background

We now give a brief description of the key characteristics of GSM as outlined in the European Telecommunications Standards Institute (ETSI) standards [ETSI92] with a focus on framing structure.

GSM uses a combined TDMA/FDMA structure. The time division is effected by accommodating either eight (at Full rate, 22.8 kbit/s) or sixteen (at Half rate, 11.4 kbit/s) timeslots-per-frame to support speech and data transmission. GSM is also a frequency division scheme, as each frame belongs to a given carrier. Each carrier has a bandwidth of 200kHz. The 890-915 MHz band is used for mobile transmission while 935-960 MHz is used for base station transmission.

Hence at full rate users may transmit or receive in the respective bands in every 8th timeslot, each of which is of 0.57 ms duration. The time compression associated with putting eight users into one radio frequency is an inherent requirement for faster base station transceiver equipment.

[PADG95] explains that the choice of a carrier bandwidth, as large as 200kHz, is made due to the negative effects of time compression. Namely, the time compression of the user data (22.8 kbit/s including error correction and coding at full rate) by a factor of around eight (or sixteen at half rate) implies a bandwidth expansion of the signal by a corresponding factor. This greatly affects the fading of the received signal. The presence of reflectors, like mountains, hills, buildings, and others, leads to many echoes. In a narrowband system, the resulting signal paths cannot be resolved in time. With a bandwidth of 200kHz, on the other hand, some degree of resolution becomes possible.

2.1.2 Half Rate Channel Standards Development

In the autumn of 1988, ETSI decided to form a new group (Traffic CHannel Half rate Speech, or TCH-HS) with the aim of creating a task force which would define the half rate channel of the GSM [MONT95]. The group was also given a specific time-frame (by 1995 at the latest) to produce a Recommendation on a speech coding algorithm for half rate speech channels, suitable for the implementation in GSM [USAI95].

The main activity of the TCH-HS was the definition of the half rate channel requirements in terms of speech quality, complexity and delay. These requisites were defined taking into account the possible evolution of speech coding technology in the time span of the group's activity. The standardisation of the full rate channel was nearly complete at the time of the TCH-HS group's inception, and therefore some margin for slightly lower performance was left to take into account the lower bit rate of the half rate channel. The three half rate channel requirements, in summarised form and as specified by the TCH-HS, are outlined below:

- ***Speech Quality:*** It was recognised that, due to the lower bit rate of the half rate channel, in each individual condition the half rate channel performance can differ significantly from that of the full rate. Two particularly problematic situations, where half rate performance degrades considerably relative to full rate performance, are: (1) Mobile to mobile tandeming, where two half rate users communicate; (2) Background audio noise, such as office and vehicular/traffic noise. Overall, the TCH-HS decided that, even taking these situations of extremely low levels of performance into account, the half rate channel should provide an average speech quality comparable with the full rate one [MONT95].

- ***Complexity:*** The foreseeable evolution of the ever improving Very Large Scale Integration (VLSI) technology, in particular Digital Signal Processing (DSP) was taken into account when fixing the complexity requirement. The TCH-HS deemed that a half rate codec with a complexity four times the full rate one could be implemented at the time of standardisation.

- ***Delay***: The GSM half rate channel delay is not only dependent on the characteristics of the codec but also on the interleaving depth and the type of mapping of half rate frames onto the super-frame, and the particular multiplexing scheme adopted in the Abis interface. The technical details regarding the above

three interleaving, mapping and multiplexing considerations are given in Recommendation 05.02 [SMG91/92]. The overall delay can be divided into *system delays* (duration of the speech frame plus the interpolation window, the transmission delay on the Abis interface and *processing delays* (delay of speech encoder and decoder parts). The TCH-HS group decided that, taking all possibilities into account, the GSM half rate channel delay could range from 190.8ms to 200ms. As the full rate delay is 188.5ms, it seems that even in the worst case (200ms) the extra 13ms of delay would not degrade transmission quality, relative to that of the full rate channel.

From these official ETSI half rate channel requirements, we can draw an interesting conclusion. The latter two requirements would seem to indicate that half rate codecs will give almost identical delay with a complexity factor (4) which is reasonable, given the ever-advancing state of the art in DSP/VLSI technology. However, some latitude was clearly left for the manufacturers when it came to setting the speech quality half rate standard to be *comparable* to that of the full rate speech quality. [MONT95] explains that the TCH-HS's definition of comparable, is that the speech quality (averaged over many diverse situations including noise and tandeming) must be

$$Quality'_{half\_rate} - Quality'_{full\_rate} >= -1dB \quad \text{... (2.1)}$$

where $Quality'_{xxx}$, *(xxx = half_ or full_rate)* is the average measure of speech quality, defined as the "signal to multiplicative noise ratio of speech degraded with multiplicative noise, by means of using the Modulated Noise Reference Unit (MNRU)", as detailed in [MONT95]. The difference of -1dB implies that if we were to linearly characterise speech quality, if full rate were to score 100%, half rate would receive only 79.4%. Since the complexity and delay requirements do not seem likely to pose difficulties, the constraint on half rate performance is imposed by this lower bound on speech quality.

It remains to be seen whether manufacturers will risk exposing customers to potentially unsatisfactory half rate codec technology only providing, say, 79.4% of the full rate speech quality, which is itself under attack in given situations. On the other hand, given the other two requirements can be met without difficulty, an increased effort to provide a half rate codec with speech quality significantly better

than this figure from the TCH-HS group's study, could pave the way for a reasonably fast and widespread customer acceptance of the half rate channel in GSM, and thus necessitate the kind of study being undertaken in this thesis.

2.2 The Model

In a simplified model, each frequency (frame) can support exactly eight full rate calls, with the *broadcast channel* ignored. This assumption is the basis for our initial 'one carrier, no broadcast functionality' model. In a real situation however, at least one timeslot (a timeslot can be used for either one full rate or two half rate channels) within each cell must be reserved for the broadcast function, meaning that at most $8n-1$ timeslots (referred to from this point on as just *slots*) are available for user traffic in an n carrier cell.

Although the GSM channel structure will often require more than just one control channel, it can be assumed that one is close enough to an average value for control channel useage. It can also be stated that this assumption does not impact on the teletraffic analysis of the problem. Hence we will also consider a more realistic model of a cell with $n=2$ and $n=3$ carriers, and hence 15 and 23 available user-traffic slots, respectively.

What is of interest to us, in this thesis, is the behaviour of half rate calls within such a framing structure. Full rate speech calls use one slot in every frame, while half rate calls will use a single TDMA slot every second frame on average.

It is expected that future mobile terminals will have the capability to operate at both rates. However, it is reasonable to assume that if a cell is equipped with half rate capability the terminal will operate in the half rate mode in that cell. The exact time organisation for slots for full and half rate transmission is summarised in Section 4.2.2.1 [MOUL92] and is in accordance with GSM standards development [ETSI92]. Mixing full and half rate traffic in a frame can result in eight full rate calls, 16 half rate calls, or any feasible combination. This is illustrated in Figure 2.1.

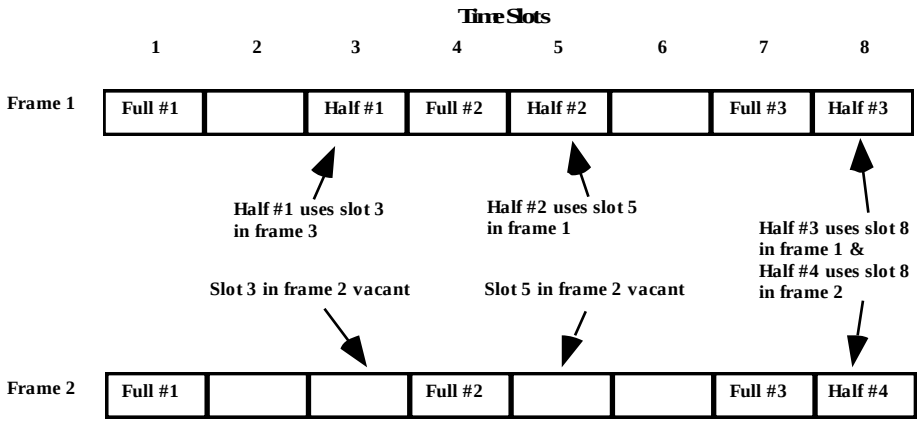

Figure 2.1: A Typical Half and Full Rate Traffic Mix

For the purpose of evaluating the traffic performance of a mixed full and half rate network, the arrangement in Figure 1 can be seen as slots being capable of supporting one full rate call or two half rate calls (refer to Figure 2.2). Note that the boundaries between the eight slots are considered *stone walls*, and in that sense, a full rate call may never be placed across one of these walls.

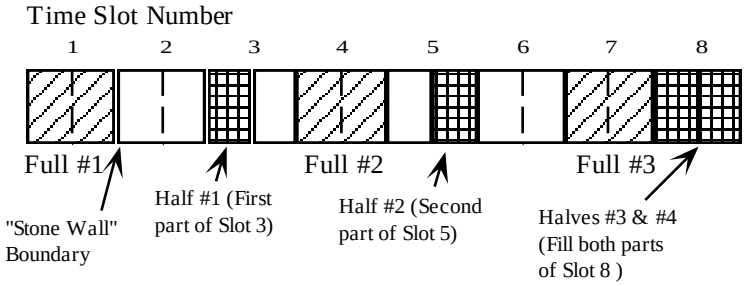

Figure 2.2: A Model for Analysing the Traffic Mix in Figure 2.1

At first glance, it might appear that introducing half rate calls will double the capacity of a GSM network. However, this optimistic view fails to consider two important factors: (1) although a half rate call uses on average half a slot, its *effective usage* is higher because even under the most efficient packing scheme, when a slot is occupied by a half rate call, no full rate call can use that slot. In this case, from the point of view of the full rate traffic, a half rate call effectively takes up an entire slot; (2) The portion of existing GSM customers that will change over to the new half rate system is unknown and may be influenced by the cost of buying a new handset and natural resistance to change.

With this notion of mixed traffic the issue is the slot allocation scheme adopted when full and half rate call arrivals and departures are registered. That is, how should we store the full and half rate calls (into empty slots and holes) as they arrive, or how should we reorganise them as they leave. Note the definitions of the above mentioned bracketed terms: (i) an *empty slot* is just that - a slot which has no full rate or half rate calls; (ii) a *hole* is the empty portion of a slot which has one half rate call occupying it. The occupying half rate call is referred to as the *isolated half*. Although some perform far better than others, the purpose of all of the allocation policies should be to attempt to allocate resources in such a way that the blocking probabilities of half and full rate calls are equalised as far as possible, and at the same time kept to a minimum. The value of blocking probability (e.g. 2%) will be referred to as the Grade of Service (GOS). The best scheme will provide the maximum number of customers per cell - *capacity*, while satisfying the abovementioned GOS constraints of equality and minimality.

2.3 Additional Related Teletraffic Issues in Mobile and Broadband Networks

The slot allocation problem discussed here in the context of GSM is related to the problem of Connection Admission Control (CAC) in multiservice networks [GUER91, HUI89, KELL91]. In particular the reservation schemes outlined in Section 3 are discussed in [RITT94, TRAN93].

In the domain of mobile network teletraffic research, particularly relevant examples are given in [CALL95] and [KWON95] where the concept of 'handover load reserve' is described as the need to set a higher priority for handover requests and at each base-station to reserve a fixed or variable number of channels for use solely by calls being handed over from one cell to another. This then guarantees a lower probability of blocking for handover requests than for new call arrivals. As will be explained in Section 3, three types of Repacking schemes involve this same concept of reservation in order to control the blocking probabilities of different types of traffic.

The repacking of half rate calls between channels is used in this work to achieve a more compressed frame structure with fewer half-empty slots and hence reduce the blocking probability for full rate calls. The generic concept of repacking active calls, often referred to as intra-cell handover, stems from early work by Cox and Reudink [COX73] which looked at 'dynamic channel reassignment' as a means of significantly increasing channel occupancy (hence capacity) for a fixed probability of blocking.

When it became apparent that the increase in capacity, required to keep up with the growing demand for mobile services, could not be achieved by progressively more complex dynamic channel assignment and reassignment algorithms, research efforts began to shift towards microcellular architectures. It was within the context of microcellular environments where more recent work involving repacking strategies has appeared, and the focus has been mainly to investigate the possibility of adaptive channel reassignment based on continual measurements of the radio connection quality.

Beck and Panzer [BECK88] describe an efficient algorithm which initiates an intra-cell handover when the reception quality of a call using a given channel decreases below a given threshold. The algorithm is efficient in the sense that it achieves 'stability' - the actual probability of calls that need a mid-conversation repacking is maintained at a very low value. In Subsection 7.4 a similar study of intra-cell handover stability is carried out for the schemes of the Repacking family.

There is also similarity between the dynamic channel assignment schemes in the work of Sivarajan et al [SIVA90] and some of the schemes which shall be described in Section 3. For example in [SIVA90] the 'Simple' scheme assigns all incoming calls to the least available frequency, much like the First Fit scheme detailed in Section 3 and first proposed by Knuth [KNUT73]. Furthermore, the scheme described as 'Maxavail' in [SIVA90] assigns to an incoming call the frequency which maximises the total number of channels available in the entire system. Although not entirely identical, such a scheme closely resembles the Best Fit scheme described in Section 3. Best Fit, first proposed by Knuth [KNUT73], maximises the number of entirely empty slots by grouping half rate calls together upon admission.

3. SLOT ALLOCATION SCHEMES

In this section we describe the nine schemes of allocating slots to full and half rate calls. Each of these schemes, except for Random, is an access control method which allocates resources in the form of slots to two types of calls. The motivation for investigating various access control schemes becomes obvious when one considers that without any form of access control in a shared resource environment [KAUF81], calls requiring a larger capacity will experience many times the blocking probability of those with smaller capacity requirements. This will be demonstrated in Section 7 by comparing the performance of the Random scheme with that of schemes where access control is imposed.

It can be said that all of the schemes are well known except the three variations of ordinary Repacking: (i) Repacking with Random Reservation; (ii) Repacking with Perpetual Reservation (of a full slot) and (iii) Repacking with Perpetual Half Slot Reservation. In Section 7, after a comprehensive analysis of the results, each scheme is allocated a mark out of 10 for implementation simplicity (10 is the most simple to implement), fairness (10 is the most fair) and efficiency (10 signifies best utilisation of network resources while keeping to a pre-defined GOS).

Implementation simplicity is the inverse of scheme complexity, with regard to the number of algorithmic rules which would need to be implemented in hardware. For example the Random scheme is much more simple to implement than any scheme from the Repacking family, because it does not have any of the rules associated with call repacking. It is also important to precisely define what is meant by the concept of scheme *fairness.*

$$f = \log_{10} \frac{P(Blocking)_{fulls}}{P(Blocking)_{halves}} \qquad \text{... (3.1)}$$

As equation 3.1 illustrates, fairness is the measure of numerical proximity of the blocking probabilities for the two types of calls, namely full and half rate. When the *f* value according to equation 3.1 is zero, we have exactly equal blocking probabilities for both types of call traffic and it can be said that such a scheme is *completely fair*. Negative *f* values are unfair to half rate calls, while positive *f* values are unfair to full rate calls.

Finally, *efficiency* is the measure of the total offered arrival rate which does NOT exceed the system's GOS.

3.1 Random

Although termed a "scheme", this method is not a true scheme; rather it is a pure "dice throwing" exercise. Namely, as full and half rate calls arrive, the task of the processor is to:

(a) In the case of full rate calls, choose at random one of the free slots available, being careful to keep within the "Stone Wall" boundaries.

(b) In the case of half rate calls, choose at random one of the free *holes*. Because of their size, these calls can "see" two empty slots in each ordinary empty slot, and thus we refer to them occupying at random the first half slot which is available. Clearly the wall boundaries do not apply here.

Finally, as each call leaves, it leaves either an empty slot or a hole, and there is no action taken by the processor at this point. Holes may be created by (i) the departure of one of two half rate occupants within a slot, or (ii) the arrival of a half rate call into an empty slot. Note that because holes must co-exist with isolated halves, the creation or elimination of one means the same happens to the other. On the other hand, empty slots will remain behind when either a full rate call departs, or when an isolated half rate call departs.

3.2 First Fit

The policy which is simplest to implement, (apart from Random) is First Fit [KNUT73, ZUKE88, ZUKE89, COFF85]. Under this policy, the eight slots in the frame are permanently allocated ID numbers (say 0-7). In the case of our system, each slot may contain (i) no calls {0}, (ii) a half rate call {1}, (iii) two half rate calls {2} or (iv) a full rate call {3}. To each of these possibilities, the value in the braces is attached. The allocation scheme then functions as follows:

(a) Each incoming half rate call is allocated to the smallest ID-number slot (i.e. imaginary packing from the left hand side of the frame), with only 0 or 1 half rate calls currently there.

(b) Each incoming full rate call is allocated to the smallest ID-number slot that is empty.

Similarly to Random, as each call leaves, it leaves an empty slot or a hole, and there is no action taken by the processor at that point.

3.3 Best Fit [KNUT73]

This scheme is more efficient than either of the two previous ones, because it specifically targets the holes that are present in the frame, and eliminates these by adding to them any new incoming half rate calls [KNUT73, ZUKE88, ZUKE89]. In this way we have more closely packed frames, freeing up room for the full rate calls. Specifically, the placement scheme is as follows:

(a) Incoming full rate calls are packed from the left of the frame, exactly as described in First Fit: allocation to the smallest ID numbered slot which is currently empty.

(b) An incoming half rate call is allocated to a hole. If more than one hole is available, the one belonging to the slot with the smallest ID number is filled. On the other hand, if none are available, we place it in the smallest ID numbered slot which is currently empty.

Once again, no action is taken upon call departure, and a hole remains behind.

3.4 Repacking [ZUKE88, ZUKE89]

This is one of the most efficient methods of slot allocation, and is almost identical to Best Fit with one major difference: the action taken by the processor when a call leaves. Namely, upon call departure,

(a) When a full rate call departs leaving an empty slot, no action is taken.

(b) When a half rate call departs, either an empty slot or a hole will remain. In the former case, no action is taken. In the latter case, if another isolated half is available, it will be moved into the hole. If not, no action is taken.

Implementation of the repacking strategy makes use of intracell handover including repacking across different radio frequency carriers within the same cell. In the

numerical examples of this research work we have considered both repacking within a single radio frequency carrier, as well as between multiple carriers.

The reader should note that a large number of intracell handovers during a call will have a negative effect on the quality of service as perceived by the customer. It is therefore important to have this number as small as possible. To reduce the number of intracell handovers, step (b) above will not be performed upon a **departure**. Instead, intracell handovers (repacking of two isolated half rate calls) will only be performed upon an **arrival** of a full rate call, given that there are no empty slots, and that there are at least two isolated halves. Although this results in a slightly different scheme at the physical layer, it does not in any way change the complexity or size of the state space or the state transition diagram, as will be explained in Section 4. We now define and clarify the above concepts.

The *state* of the system is a two or three dimensional vector, depending on the scheme in question, with the first element representing the number of full rate calls and the second element representing the number of half rate calls in the system. In those schemes that require it, the third element of the vector represents the number of isolated halves in the system. The *state space* is the finite set of all feasible states. A *state transition diagram*, also known as a Markov chain, shows transitions between states as probability weighted links. Such a diagram is also a representation of the steady state equations, which when solved yield the steady state probabilities of the system being in any given state.

The result of such a scheme is that a full rate call arrival will not find a situation where there is more than one hole (when there is sufficient demand to warrant repacking). Therefore, we expect an increase in ***utilisation*** (*where utilisation is defined as the average number of occupied slots divided by the total number of slots*). The only essential difference between admittance criteria for full and half rate calls is that when there are seven and a half slots filled, a half rate call will be admitted, whereas a full rate call will not.

3.5 Repacking with Perpetual Reservation (RPR)

This scheme is a variant of Repacking. The only difference in the scheme, is that the entire last available slot must always be kept *reserved for use only by full rate calls*. This is not necessarily always the same **physical** slot; rather, it is always the **last free**

slot, whichever one it may be. Therefore, the admission of full rate calls stays identical, whereas the admission of half rate calls is purposely impeded, in the sense that they can only ever make use of seven of the eight slots of the frame. The reason behind this is equalisation, since it is expected that with ordinary Repacking, the blocking probabilities for half rate calls tend to be far smaller than those of full rate calls.

3.6 Repacking with Perpetual Half Slot Reservation (RPHSR)

Like RPR, this scheme is another variant of ordinary Repacking. It was first proposed in [TRAN93]. We reserve half a slot thus introducing a small amount of resource wastage, and basically "forbidding" the half rate calls to enter the system when only one half slot remains available. In this way, the blocking probability of full and half rate traffic is equal for every traffic mix at the expense of lowering the half rate call utilisation somewhat. This scheme is also discussed in [RITT94, TRAN93] as an example of a blocking probability balancing mechanism in a CAC context for Asynchronous Transfer Mode (ATM) networks.

For instance in a transmission system with total capacity for eight basic bandwidth units (BBUs) the *threshold*, according to [RITT94, TRAN93], is set at eight minus the number of BBUs required by the call type with the largest BBU requirement. For example, if we only have two call types, and the type with the largest requirement uses one BBU the threshold would equal seven BBUs. If at any stage more BBUs than the threshold value are occupied, either type of call arrival is blocked. This guarantees that the same proportion of arriving calls (not necessarily absolute numbers) will be blocked, for both call types. This idea of blocking probability equalisation can easily be extended to a multi-service system with many different bandwidth requirements.

3.7 Repacking with Random Reservation (RRR)

Once again, this is a variation of the Repacking scheme, and bears a lot of resemblance to the Perpetual Reservation scheme. Unlike RPR or RPHSR, where either an empty reserved slot or half an empty reserved slot is always put aside (i.e. **deterministic, perpetual reservation**), the Random Reservation scheme states that this reserved slot is only put aside some of the time. In other words, if only one

empty slot remains and a half rate call arrives, then with probability p_1, the call is accepted, and with probability $(1-p_1)$ the call is rejected. If only a half of an empty slot remains is available then with probability p_2, the call is accepted, and with probability $(1-p_2)$ the call is rejected.

The probability p_1 is chosen as a function of the arrival rates for the two types of traffic, λ_1 and λ_2, $p_1 = f(\lambda_1, \lambda_2)$. It is logical that, if perchance the full rate arrivals were an order of 10 or 100 smaller than half rate arrivals, it would be most beneficial to have a high value of p_1.

p_2 on the other hand is associated with the event of already having admitted a half rate call into the reserved slot, and therefore it should be chosen as a function of not only the arrival rates, but also of the current number of half rate calls in the system. That is $p_2 = f(\lambda_1, \lambda_2, N_h)$. Intuitively, if we have many half rate calls in the system and a half rate call attempts to fill up the reserved slot, we may reject it because:

(i) There are many other half rate calls, meaning an increased probability of departure elsewhere, so possible success on a retry.

(ii) Since half rate calls have "run of the system" in such a case, in order to reduce the difference in the GOS for both types of customers (i.e. force *f* towards 0), it is advisable to wait for the lone half within the reserved slot to depart, and thus free some semi-prioritised space for the full rate customers.

However, for simplicity in this work it was assumed that p_2 = 1.0 (i.e. if the reserved slot has a half already in it, ALWAYS fill it up). Interestingly, the special case of p_1=1.0 and p_2 = 0 has already been mentioned above in Subsection 3.6 - it is nothing more than the case of forcefully preventing the half from entering the system when it has $(8n-1)-½$ half slots full, with a view to equalisation of blocking probability *(RPHSR)*. On the other hand, the case of p_1= 0 is clearly the case of Repacking with Perpetual Reservation, discussed in Subsection 3.5. Using the eight slot frame, and testing the RRR scheme with a number of values of p_1, the aim was then to empirically find the optimal p_1, which would ensure that fairness in blocking probabilities for both types of customer was achieved. Clearly, a particular value of p_1 cannot be expected to satisfy every conceivable traffic mix at every conceivable utilisation. Therefore, it was not surprising that the values of p_1 which satisfied this fairness criterion were in the range $0.5 < p_1 < 0.7$, for a very wide range of traffic mixes ($0.1 < \frac{\lambda_1}{\lambda_1 + \lambda_2} < 0.9$). It was therefore decided to take the middle of this range

and set p_1 = 0.6 for this scheme, regardless of traffic mix, utilisation or frame size. As results in Section 7 show, this choice turned out to be almost truly optimal because it promoted the RRR scheme to first place in efficiency and a very close second to RPHSR in fairness.

We now describe two methods which are not very efficient, but may be considered by engineers because of their simplicity in implementation.

3.8 Fixed Boundary Reservation

An imaginary wall is imposed at some pre-defined point within the frame. On one side of this boundary, the slots may only be used by full rate calls, while on the other, only half rate calls may be accepted. It is immediately apparent that this option is simple to implement, as it never mixes the two types of traffic. However, the utilisation of such a scheme is intuitively poor, due to resource wastage. Unlike the other schemes, this allocation technique is described by the M/M/N/N queue. The Kendall notation used here [KLEI75] denotes a queue with a Poisson arrival process, a negative exponential service time distribution, and at most N available servers. This is a 'blocked calls cleared' situation, where each newly arriving call is given its private server. If however a call arrives when all N servers are busy, that call is lost. The probability of this event is well known in literature as *Erlang's loss formula* or the *Erlang-B formula*.

In the specific problem of the Fixed Boundary Reservation scheme, N would be equal to the *number of slots available to incoming calls* on a certain side of the boundary. So if we have an eight slot frame, and divide the resources equally, full rate calls will receive service from an M/M/4/4 queue, while half rate calls, being able to fit two-to-a-slot, will be served in an M/M/8/8 queue. The blocking probability may then easily be obtained by use of a simple recursive version of the Erlang-B formula [HARR82].

3.9 Sliding Boundary Reservation

Unlike the previous scheme, the sliding boundary allocation scheme has to periodically make a decision on how to split the eight slots between the two types of traffic (i.e. it is not pre-defined). Although the term 'sliding' may be sometimes used

in literature to imply that one type of traffic can cross over the boundary, what is described here is essentially a scheme where the boundary cannot be crossed by either type of traffic, with the boundary location chosen at set periodic intervals according to the measured traffic mix. Given that the arrival rate of full rate calls is λ_1, and half rate calls is λ_2, the subdivision is made in the following way:

$$\textit{No. of Full Slots Available to Full Rate Calls} = \text{round}\left[8*\frac{\lambda_1}{\lambda_1+\lambda_2}\right] \quad ...(3.2)$$

$$\textit{No. of Full Slots Available to Half Rate Calls} = \text{round}\left[8*\frac{\lambda_2}{\lambda_1+\lambda_2}\right] \quad ...(3.3)$$

Above, the "round" function performs a rounding of a real quantity to the nearest integer (e.g. 2.345 to 2). The end result is a rather rough matching of the ratio of available reserved slots and the ratio of traffic intensities. Although rough, this method at least attempts to work on the principle of demand and supply, and one would expect less resource wasting.

4. SYNTHESIS OF STATE TRANSITION DIAGRAMS

One way of solving this problem is to consider a set of as many state variables as there are slots, each of which could take four values. This would result in a state space which described *every possible permutation* of half and full rate caller occupancies in the available slot. As a result, any such state space would be enormous, especially for realistic numbers of time slots. However, it is possible to reduce the state space to a manageable size by using some ideas from [ZUKE89], in a mapping approach similar to that proposed by Kaufman [KAUF81] and Roberts [ROBE81]. The only difference is that in our problem, the mapping is from an *m* dimensional state space down to a 2 or 3 dimensional state space, rather than a one dimensional space, as in [KAUF81, ROBE81].

Using the model defined in Section 2 as a framework, all of the allocation schemes except First Fit, may be described by such a reduced state space where each state is a vector with the elements: (1) ***i***, the number of full rate calls currently in the frame; (2) ***j***, the total number of half rate calls; and only for Random and Best Fit schemes, (3) ***k***, the number of isolated half rate calls in the frame.

The last parameter is only needed for the Random and Best Fit allocation schemes, because unlike the others, these schemes can have more than one isolated half rate call. For example, the nature of the scheme behind Repacking, guarantees that a valid state (4,6) means four full rate calls packed into four slots, and six half rate calls perfectly compressed into three slots. With Random and Best Fit allocation however, having six half rate calls could mean any feasible scattered arrangements within the frame.

First Fit is not included in this type of state evaluation by analytic numerical methods, because it was not possible to perform the state space reduction mapping. The reason lies behind the fact that unlike the other methods, each slot is numbered, and as such, there must be the same number of state parameters as there are slots. Two or three parameters could not adequately provide a state space description. It was therefore sensible to only simulate this allocation scheme.

The Fixed and Sliding Boundary allocation techniques are both described by the M/M/N/N queue, where N is simply the number of slots available to incoming calls on a certain side of the boundary. Hence, the blocking probability is easily obtained by use of the Erlang-B formula, so neither state evaluation or simulation is required.

On the basis of the model in Section 2, and for the six relevant allocation schemes, we now proceed to define each one separately, by way of state transition diagrams. Because of the complexity of the entire 3-D state diagram for Random and Best Fit Schemes, in Subsections 4.1 and 4.2 we will focus on showing an example of each transition type.

4.1 Random Allocation

Figure 4.1 is a small subset of the state transition diagram for the Random allocation scheme, which has been chosen carefully so as to illustrate every type of transition characterising this scheme.

To each state is attached an extra incoming and outgoing arrow in order to reinforce the fact that this is a **subset** of the entire 3-D state transition diagram. Some state transitions have been purposely left out in Figure 4.1, for clarity, since a typical state will have six outgoing and six incoming transitions, as the total number of transition types is six. Of particular interest are the transitions weighted by probabilities. For example, when we are in the state (2,3,3), there are three empty slots as well as three holes.

Hence, because the scheme revolves around a completely randomised placement of all calls into the frame, the probability of the Hole Elimination transition taking place must be (3/9) while that of the Hole Creation transition (6/9).

Another example is the case where the frame is in the (2,4,2) state and a half rate caller departs. Once again, the departure is a random process. The frame contains two isolated halves (and hence holes), as well as two halves packed together in one slot. Hence, it is intuitive that for half rate call departures the probabilities of the Hole Elimination and Creation transitions will be equal to (2/4).

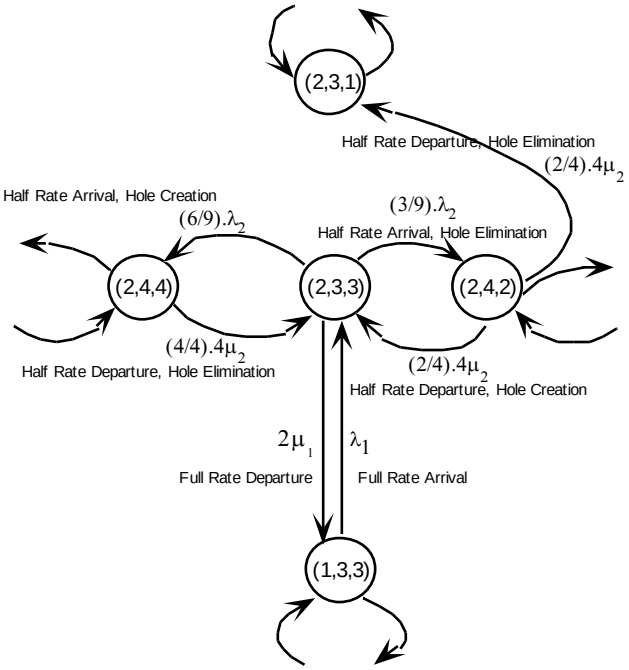

Figure 4.1: Random Allocation, Examples of Six Transition Types

4.2 Best Fit Allocation

As for the previous scheme, a characteristic subset of the Best Fit state transition diagram has been chosen in order to illustrate the five types of transitions possible.

Firstly, note that the departures are based on exactly the same random process which was mentioned in Subsection 4.1, and hence probability weighted transitions occur either creating a hole or eliminating a hole upon departure. The difference between Random and Best Fit state transition diagrams is evident in the transitions representing the arrival process.

Namely, the Best Fit scheme specifically targets any available holes to be filled, therefore unlike in Random allocation, Figure 4.1, where two probability weighted half rate arrival transitions could occur, only one type of transition resulting from a half rate arrival is possible in Figure 4.2 below. As a result, this state transition diagram is characterised by only five types of transitions, as illustrated below. The example deterministic half rate arrival transitions are highlighted in bold.

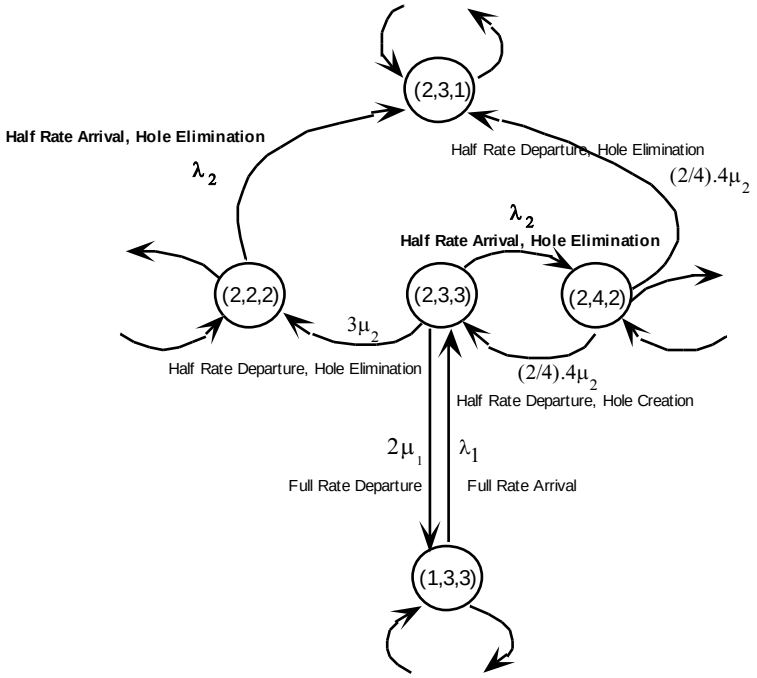

Figure 4.2: Best Fit Allocation, Examples of Five Transition Types

4.3 Repacking

Because the Repacking family of schemes has two state parameters, the resulting state spaces will be two dimensional and it is therefore possible to construct 2-D drawings, as in Figures 4.3 through to 4.6.

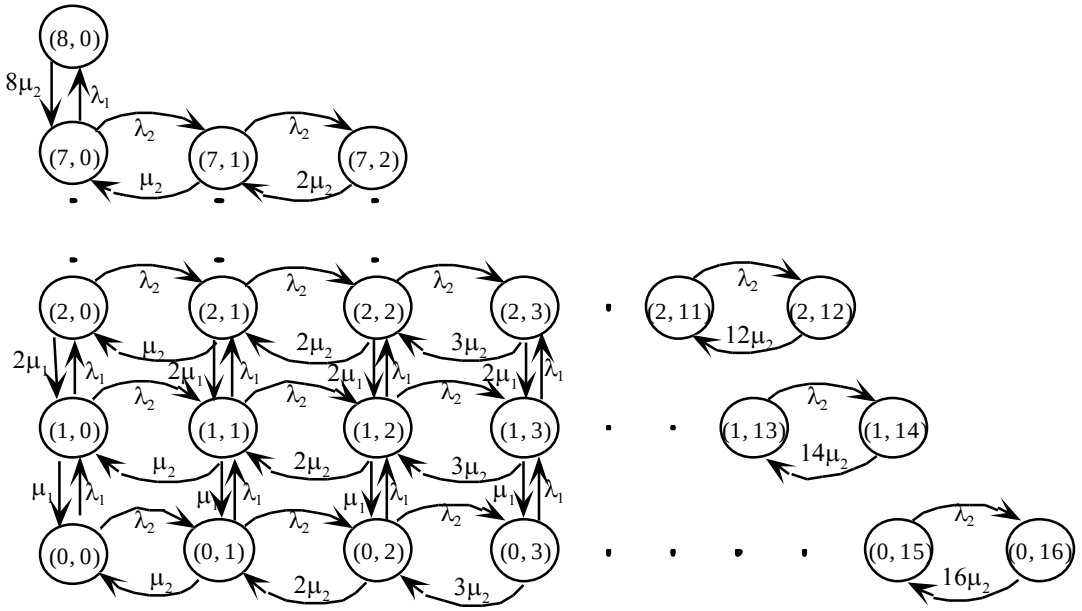

Figure 4.3: Repacking for an Eight Slot System

The examples given are each based on a generic system with eight slots per frame; note that the state spaces have been only partially drawn (size is very large) to illustrate their characteristic features (with any irregularities highlighted or shaded). We start with the simplest of the Repacking family of algorithms.

4.4 Repacking with Perpetual Reservation (RPR)

The state space is similar to that of Repacking, with the major exception being that the last available slot must always be reserved exclusively for full rate callers. That is why the diagram is identical to Figure 4.3, except that states (0,15) and (0,16) are unreachable and hence shaded. Also, let us define *full states* as those where either (i) all eight timeslots are occupied, OR (ii) seven and a half slots are occupied and one half slot hole exists.

Examples are (2,11) and (2,12). As a result of the modified admission scheme, the diagram shows that such states can now only be reached if the transition into them is a **full rate arrival.** This is evident in Figure 4.4 due to the absence of horizontal transitions to such full states.

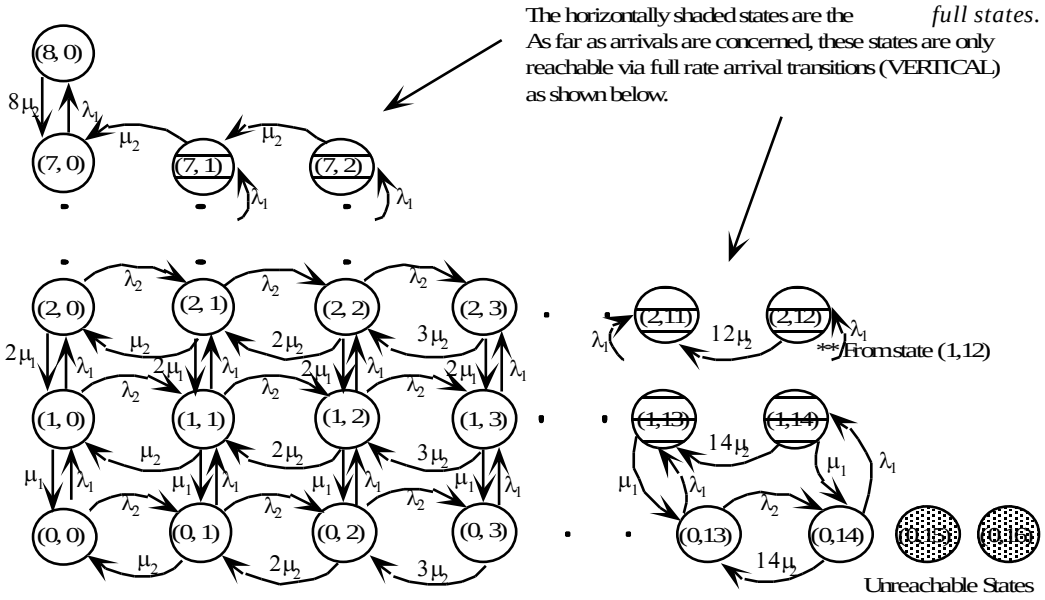

Figure 4.4: Repacking with Perpetual Reservation for an Eight Slot System

4.5 Repacking with Perpetual Half Slot Reservation (RPHSR)

This scheme is almost identical to RPR, except that there are less *full states* since they are now defined as only those states where all eight slots are occupied.

As in the case of the RPR scheme, these full states are inaccessible by horizontal transitions. In other words, they cannot be reached by half rate arrival transitions - only by full rate arrival transitions. Since there are less of these states which are inaccessible by half rate arrivals, and given that the state (0,15) is reachable unlike in the RPR scheme, intuition suggests that half rate customer utilisation should be improved for this scheme.

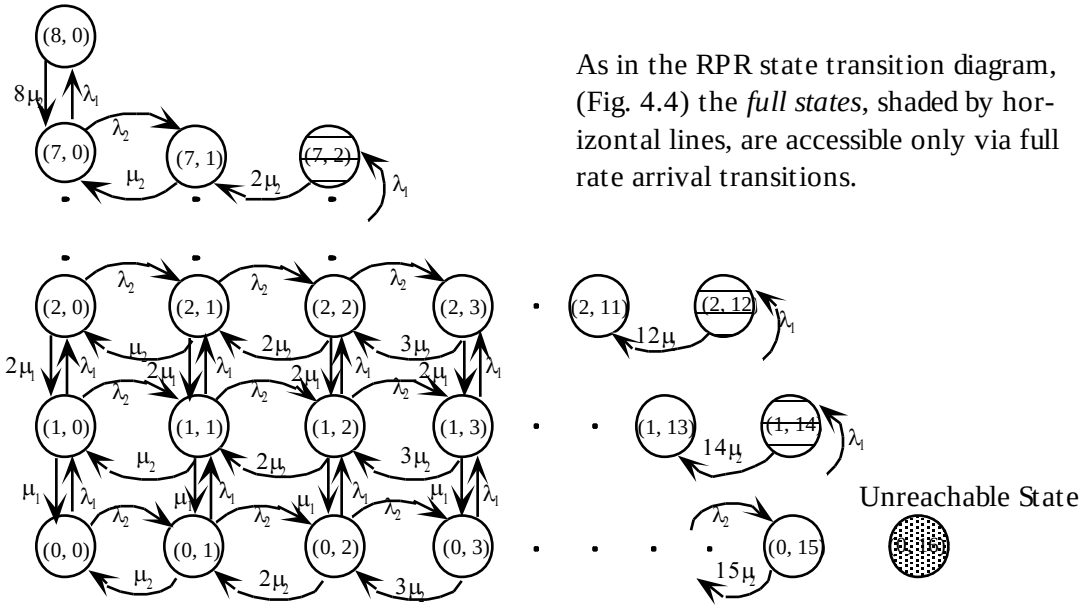

Figure 4.5: Repacking with Perpetual Half Slot Reservation for an Eight Slot System

4.6 Repacking with Random Reservation (RRR)

Figure 4.6 illustrates the concept of random reservation. Given that a half rate call arrives, the transition into one of the shaded states will occur (i) with probability p_1 if an entire empty slot is available (e.g. transition from state (7,0) to (7,1)), and (ii) with probability p_2 if only half a slot is available (e.g. transition from state (1,13) to (1,14)). All these shaded states illustrate the *non-deterministic filling up of the last available slot by half rate callers*. The rate of entering each shaded state is given either by $p_1\lambda_2$ or $p_2\lambda_2$.

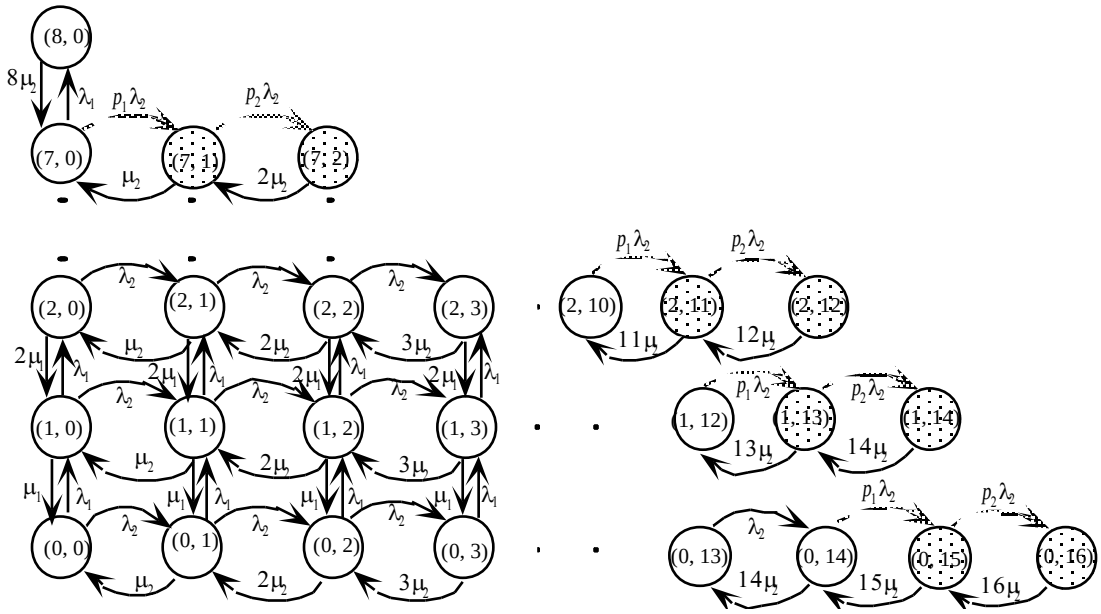

Figure 4.6: Repacking with Random Reservation for an Eight Slot System

The family of repacking schemes, as described in the previous section, may be implemented at the hardware level so that either (a) every departure may *potentially* prompt a repacking, or so that (b) only full rate arrivals which *cannot fit* into the frame prompt a repacking. The latter scheme minimises the overall number of repackings, in an attempt to reduce the number of 250-500 ms *silent* or *click* periods which result from intracell handovers. However, from the point of view of state space complexity, the actual hardware level implementation has no impact on the size of the valid state space (the drawings in Figures 4.3 - 4.6 stay identical). This is because there are only two state parameters - the number of fulls and number of halves, in the frame. As a result, the frame can have the halves distributed in any fashion, under the condition that when the need arises maximal packing compression will be effected. As far as blocking probability is concerned, this is equivalent to performing the repacking upon every half rate departure.

5. GENERATION AND NUMERICAL SOLUTION OF STEADY STATE EQUATIONS

The analytic method was based on generating the state transition rate matrix, *Q*, and solving the related set of equations using a Gauss-Seidel iteration technique. The focus in this section is on the way such a large matrix is generated, and various computational aspects, including limitations. As outlined in [HUEB93] and [RITT94] it would seem that rather than iteratively solving a large number of steady state equations, this type of problem is amenable to solution using the exact product-form (and hence Kaufman-Roberts recursion [KAUF81, ROBE81]) method, as the number of traffic classes is sufficiently small (two classes).

However, closer examination of the state transition diagrams presented in Section 4 reveals that pure Repacking is the **only** scheme for which the product-form equations could be used. This is the case because, for example, in the Repacking state (3,7) we can *always say* that the rates of departure are: (i) $3\mu_{full}$ for full rate calls and (ii) $7\mu_{half}$ for half rate calls respectively, independent of any other factors. On the other hand, the other three Repacking family members, and Random and Best Fit allocation, all either have specific weighting factors associated with particular transitions, giving a 'non-standard' state transition diagram, or they map two traffic classes to a 3D state space; in either case, it makes it impossible to use the exact recursive or product-form solutions.

Taking the Random state space as an example, the departure rates from the state (3,7,3) are $3\mu_{full}$ as before for full rate calls; however, the half rate call departures are complicated by the completely random possibility of either an isolated or non-isolated half rate call departing. This phenomenon, along with the way each particular allocation scheme requires certain states to be unreachable or only entered with certain probabilities, makes it impossible to use the exact product-form solution.

5.1 Transition Rate Matrix Generation

The state parameters are: ***i*** = number of full rate calls ("fulls") present; ***j*** = total number of half rate calls ("halves") present; ***k*** = number of isolated half rate calls present in the frame. As usual, a_x, μ_x are the arrival/departure rates, with ***x*** = 1 for full rate and ***x*** = 2 for half rate calls. It should be noted that the matrix generation

algorithm, presented in four steps below, specifically considers valid states only, eliminating all *(i, j, k)* points not satisfying this requirement.

Step 1

Obtain the numerical difference in the state parameter indices.

$$\Delta(i,j,k) = (i',j',k') - (i,j,k) \qquad \ldots (5.1)$$

Note that the destination and source states, *(i', j', k')* and *(i, j, k)*, must always be valid.

Step 2

Determine whether the difference $\Delta(i, j, k)$ represents a **valid** state transition. If **not**, then $Q(i', j', k') = 0$, otherwise follow through to Step 3.

Step 3

From the difference determine what type of state transition event has occurred. The event then has a subset of two or one possible sub-events, based on the **(i)** slot allocation scheme, and **(ii)** the source state. Having determined the sub-event, finally record the relevant rate of transition (e.g. $Q(i', j', k') = \lambda_1$ or $Q(i', j') = \mu_2$).

Step 4

Perform *blocked state check* to see whether, for that allocation scheme, the state in question is one in which the arrivals of full or half rate calls will cause blocking.

In order to illustrate the algorithm, it is necessary to provide some examples, each looking at a different allocation scheme. Note the increased complexity associated with the Random Allocation and Best Fit schemes as opposed to the Repacking family, is due to the need for one more state parameter.

Example 1

As the first example, consider the Random Allocation scheme and a transition from state (1,5,3) to state (1,6,4). The index-difference is (0,1,1) and tells us that a half rate call has arrived, increasing the number of isolated half rate calls. This breaks down into the following event specification,

- ***Scheme:*** Random Allocation
- ***Event:*** Half Rate Call Arrival
- ***Sub-Event:*** Isolated Half Creation
- ***Q Matrix Entry:*** $Q(1, 6, 4) = (6/9)*\lambda_2$
- ***Blocked State (Destination):*** Fulls - No ; Halves - No.

The term (6/9) is a measure of the probability of the "Isolated Half Creation" sub-event occurring, given a "Half Rate Call Arrival" event occurring. The probability is 6/9 because, when we are in the state (1,5,3), out of the available nine empty half slots, six are in empty slots, and the other three are holes matched up with isolated halves. Figure 5.1 illustrates this.

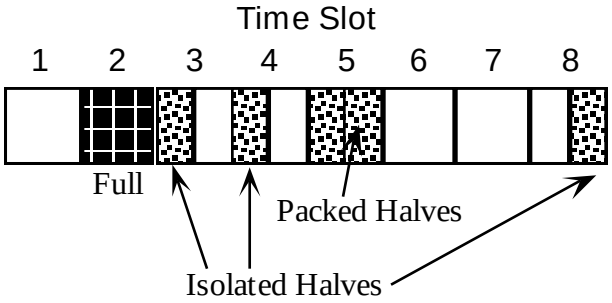

Figure 5.1: Typical Distribution of Half and Full Rate Calls in the (1,5,3) State

Example 2

The transition involving the departure of a half rate call in the Best Fit allocation scheme, is our next illustration. Figure 5.2 illustrates the call type distribution in the state (3,7,3), that is, prior to a half rate call departure. As an example let us assume that the half rate call departure will create another isolated half rate call. Hence, when the departure occurs, the state changes from (3,7,3) to (3,6,4), since one of the four half rate calls which are packed two-to-a-timeslot, departs.

- ***Scheme:*** Best Fit Allocation
- ***Event:*** Half Rate Call Departure
- ***Sub-Event:*** Isolated Half Creation
- ***Q Matrix Entry:*** $Q(3, 6, 4) = (4/7)*\mu_2$
- ***Blocked State (Destination):*** Fulls - Yes ; Halves - No.

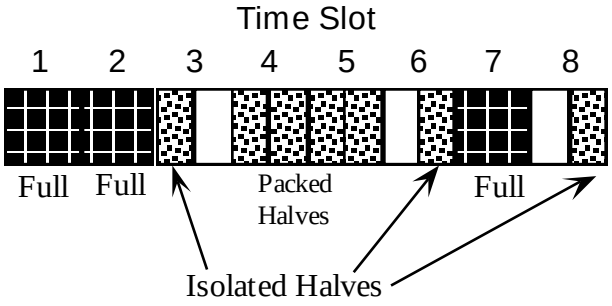

Figure 5.2: Typical Distribution of Half and Full Rate Calls in the (3,7,3) State

Example 3

Consider the Repacking scheme, and the transition from state (4,6) to state (4,7). An index difference of (0,1) tells us that the arrival of a half rate call is in question. The absence of the third state parameter means that there are no sub-events associated with events.

- ***Scheme:*** Repacking
- ***Event:*** Half Rate Call Arrival
- ***Q Matrix Entry:*** $Q(4, 7) = \lambda_2$
- ***Blocked State (Destination):*** Fulls - Yes ; Halves - No.

5.2 Matrix Manipulation

Part of a typical ***Q*** matrix is shown in Figure 5.3. Note that the given matrix is for illustration purposes only, and therefore not all of its elements are shown. This matrix has been generated for the case of a single carrier, with just one eight-slot frame. The resulting size is 160 by 160 elements, thus the reason for showing only part of it. With the matrix in the form shown in Figure 5.3, the steady state equations may be expressed as

$$\boldsymbol{pQ = 0} \qquad \text{... (5.2)}$$

(i',j',k') / (i,j,k)	(0,0,0)	(0,1,1)	. . .	(1,0,0)	(1,1,1)	. . . .	(7,0,0)	. . .	(8,0,0)
(0,0,0)	$-(\lambda_1+\lambda_2)$	λ_2		λ_1	0		0		0
(0,1,1)	μ_2	$-(\lambda_1+\lambda_2+\mu_2)$		0	λ_1		0		0
. . .									
(1,0,0)	μ_1	0		$-(\lambda_1+\lambda_2+\mu_1)$	λ_1		0		0
(1,1,1)	0	μ_1		μ_2	$-(\lambda_1+\lambda_2+\mu_1+\mu_2)$		0		0
. . .									
(7,0,0)	0	0		0	0		$-(\lambda_1+\lambda_2+7\mu_1)$		λ_1
. . .									
(8,0,0)	0	0		0	0		$8\mu_1$		$-8\mu_1$

Figure 5.3: State Transition Rate Matrix, Q, for the Random Allocation scheme

The unknown in the above equation is the ***p*** matrix, a row vector of the probabilities of state

$$\boldsymbol{p} = [p_1 p_2 \ldots . p_m] \qquad \text{... (5.3)}$$

Equation (5.2), however, is not a form practical for implementing the Gauss-Seidel iterative process [COOP81]. Hence we manipulate the elements of the original ***Q*** matrix, creating an augmented matrix, ***Q'***, and enabling the matrix equation to be written in the form

$$\boldsymbol{pQ'} = \boldsymbol{p} \qquad \text{... (5.4)}$$

The manipulation revolves around two operations: column *i* of ***Q*** is divided by $-q_{ii}$ and the *i*th diagonal element of ***Q*** is set to zero.

5.3 Numerical Solution

5.3.1 Convergence Criteria

Let us consider a general set of linear equations, following the methodology of 4.6 in [COOP81]

$$\boldsymbol{Qx} = \boldsymbol{b} \qquad \text{...(5.5)}$$

This is of the form of equation (5.2), where ***Q*** is a given square matrix, while ***b*** is a given vector (the zero vector in this case) and ***x***, the unknown vector is equivalent to

our ***p***. A sufficient condition for convergence of a Gauss-Seidel iteration scheme is that ***Q*** be *irreducible* and exhibit *weak diagonal dominance*. So if q_{ij} is the element in row *i*, column *j*, of our irreducible matrix ***Q***, then convergence will be guaranteed if

$$|q_{ii}| \geq \sum_{\substack{j=1 \\ j \neq i}}^{m} |q_{ij}| \qquad (i = 1,2, \ldots ,m) \qquad \ldots(5.6)$$

and for at least one *i* $$|q_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{m} |q_{ij}|, \qquad \ldots(5.7)$$

where *m* is the number of rows (hence columns, since we have a square matrix) of ***Q***. It is apparent by inspection of the sample ***Q*** matrix in Fig. 5.3, that this criterion for convergence is only partially met. Namely, equation (5.6) is true while equation (5.7) is not satisfied in any row. This should come as no surprise since [COOP81] explains that this criterion is not usually fully satisfied by steady state probability state equations.

Even with only equation (5.6) holding true for most probability state equations, they still have a large concentration of "mass" of the matrix ***Q*** along the main diagonal with a large number of zero elements off the main diagonal. This phenomenon is clearly evident in our sample ***Q*** matrix in Fig. 5.3.

It has been found that Gauss-Seidel iteration proves very useful in the context of solving such probability state equations, since this 'main-diagonal concentrated-mass' property makes the iteration process more likely, and faster to converge.

5.3.2 Matrix Rank

Each of the allocation scheme ***Q***-matrices, if it has for example *m* rows by *m* columns, shall be of ***rank (m-1)***. That is, each row sums to zero and any one of the columns can be derived from the other *(m-1).*

Hence, one of the equations is redundant and the other *(m-1)* equations determine the solution up to a constant factor. This factor is the normalisation condition of all state probabilities summing to one,

$$\sum_{i=1}^{m} p_i = 1 \qquad \text{...(5.8)}$$

It has been shown that, for many applications, although one equation is redundant, the convergence of the procedure is sometimes accelerated by using all the state equations. In the presented solution algorithm all the state equations are used. Furthermore, the normalisation equation, (5.8), can be applied after each complete round of iteration or after the last stage. In the solution algorithm used, the normalisation is done at the end of the iterations.

5.3.3 Computational Benefits and Limitations

With the state reduction mapping being employed, the speed of the Gauss-Seidel iterations converging to a solution (with Rel. Error < 10^{-5}) in comparison to the simulations, was as expected many times faster, and allowed us to explore performance analysis in depth. Namely, for a given GOS, of say 2%, by solving an optimisation problem, we found the maximum system capacity (defined as the number of full plus half rate call arrivals) at varying traffic mixes (0 to 100% full rate call arrivals as a proportion of all call arrivals). This will be further explained in Section 7. Because of the size of the state space, this type of analysis was *only* numerically tractable for systems of up to three carriers (frames with 3 x 8 timeslots, minus the reserved broadcast channel).

In the case of the Repacking scheme only, it was decided to use the exact recursive solution presented in [RITT94] to verify the state space enumeration method. The Kaufman-Roberts algorithm [KAUF81, ROBE81] for a two class system is of the form,

$$\tilde{p}(m) = \begin{cases} & :m = 0 \\ & :m < 0 \\ \dfrac{1}{m}\displaystyle\sum_{i=1}^{2} \tilde{p}(m - m_i) m_i \frac{\lambda_i}{\mu_i} & :0 < m \le M \end{cases} \qquad \text{... (5.9)}$$

Let ΔC represent a basic bandwidth unit (BBU) - the greatest common divisor of the individual classes' bandwidth requirements C_i, where the index i = 1 for full rate users and i = 2 for half rate users. In equation (5.9) the state denoted by m represents the total capacity used by all users, measured in the number of BBUs. m_i is the

number of BBUs required for users of class *i*. In our case, m_1 = 2 BBUs for full rate calls and m_2 = 1 BBU for half rate calls. In general the BBU requirement is determined by the use of $m_i = C_i / \Delta C$, and the maximal number of basic bandwidth units, *M*, supportable is governed by the overall capacity *C*, such that $M = \lfloor C / \Delta C \rfloor$. The brackets indicate the largest integer smaller than that quantity. In this work, half a slot is the BBU and so *M* is equal to twice the number of slots available to user traffic (i.e. *M = 2*N_Channels*).

Note that there are only *M* states in one dimension, since the users' requirements are mapped from two dimensions to one by use of the m_i variable. The state probabilities are, after normalisation,

$$p(m) = \tilde{p}(m) \cdot \left(\sum_{m=0}^{M} \tilde{p}(m) \right)^{-1} \qquad ...(5.10)$$

And finally the blocking probability B_i for class-*i* calls can be calculated as,

$$B_i = \sum_{m=M-m_i+1}^{M} p(m) \qquad ...(5.11)$$

Given that the Gauss-Seidel solution process which enumerated state (and hence blocking) probabilities converged to a relative error of $E < 10^{-d}$, by testing for d = 3, 4 .. 10, it was found that both methods of solution for Repacking yielded identical blocking probabilities to the dth decimal place.

6. THE SIMULATIONS

6.1 Simulating the Allocation Schemes

In this research, both the numerical solver programs and the simulations which verified these, had a common thread - the exploitation of the same state space. The obvious exceptions were the Fixed and Sliding Boundary schemes which did not require simulations, and the First Fit scheme, which had the requirement for the same number of state parameters as there were timeslots, without any possibility of reduction, as explained in Sections 4 and 5. As a result, only simulation of the First Fit scheme was feasible.

All of the simulations were of the Monte Carlo type and of a *discrete-event digital* nature [COOP81], where the specified average arrival and departure rates were used as parameters to random number generation subroutines. A significantly large number of discrete events caused the entire valid state space to be visited, with the probability of blocking and other useful statistics being recorded during the run. While the simulations were a good way of confirming the accuracy of analytic results, they did take up significant resources (CPU time) and it was not practical to obtain a fully blown performance analysis of the methods. For example, the required length of time to solve the optimisation problem mentioned in Subsection 5.3.3 would have been totally impractical, and a couple of orders of magnitude longer than solving it by numerical analytic means. This phenomenon was especially pronounced in the case of the First Fit simulation, due to its increased state space complexity.

6.2 The Voice Quality Impact Simulation

As the results in Section 7 will demonstrate, in terms of the overall score, one of the two best (although reasonably complex) schemes is that of Repacking with Random Reservation (RRR), since it achieves almost exactly equal blocking probabilities for half and full rate calls over a wide range of traffic mixes; it also achieves the equal best maximum customer capacity subject to a GOS constraint. It can therefore be taken to be a good representative of the Repacking family of schemes.

In order to find out the detrimental effect of repacking on voice quality, a simulation of the RRR scheme was designed with the specific aim of obtaining a quantitative measurement of *how likely a given customer's call is to be subjected to an intracell handover* (i.e. to be repacked). It was outlined in Subsection 3.4 that a large number

of intracell handovers during a call will have a negative effect on the quality of service as perceived by the customer. It is therefore important to have this number as small as possible, and hence implement all of the repacking schemes (3.4 - 3.7) in such a way so that the repacking of two isolated half slots (and hence the elimination of two holes) will only be performed upon an **arrival** of a full rate call, given that there are no empty slots, and that there are at least two isolated halves. Although this results in a slightly different scheme at the physical layer, Section 4 has shown that it does not in any way change the complexity or size of the state space.

This *Voice Quality Impact* simulation therefore uses this "repack-only-when-necessary" variant of the scheme, and like the First Fit simulation, keeps track of the exact current state of each slot - whether it is empty, or has one/two half rate calls, or a full rate call. Upon completion the simulation gives the parameter which then quantifies the impact of repacking on voice quality,

$$E[Calls\_Repacked] = (Number\_Half\ Rate\ Calls\ Repacked) / (Number\_Half\ Rate\ Calls\ Carried) \quad ...(6.1)$$

where the total number of half rate calls carried is simply the number of offered calls less the number of blocked calls. This parameter is a mean or expected value for the proportion of all carried half rate calls which are repacked. Importantly, one should note that *full rate calls are never repacked*, hence only half rate calls are considered since only they may suffer an eventual degradation of voice quality.

This Voice Quality Impact simulation is run with the **same traffic conditions** which were solved for the RRR scheme in the GOS-constrained optimisation problem. Thus, we are observing the probability of intracell handover-related voice quality degradation specifically under optimal traffic conditions and for the optimal scheme, with regards to efficiency. The fact that *this highest achievable traffic loading also represents the worst case scenario* for half rate call voice quality degradation is very useful, because it shows at what expense to quality this ideal scheme operates, if it was to operate at the desired conditions (i.e. peak utilisation).

7. RESULTS

7.1 Preliminary Investigation of Scheme Behaviour

Before performing direct quantitative comparisons between the maximum available capacity of each scheme, which is undertaken in Subsection 7.2, it is worthwhile to investigate the blocking probability of the nine schemes under varying traffic mixes. An eight-slot frame is chosen. Because the goal of this Subsection is to explore the general behaviour patterns of the blocking vs. traffic-mix curves, we do not consider here specific systems of say, one or three carriers, with reserved broadcast channels (i.e. 7 or 23 slot frames, respectively). Such multiple carrier frequency systems are investigated in Subsections 7.2 - 7.4.

Figures 7.1 to 7.9, display for each scheme, the full and half rate call blocking probabilities against the proportion of full rate calls arriving, denoted by

$$P_{Full} = \frac{\lambda_1}{\lambda_1 + \lambda_2} \qquad \text{... (7.1)}$$

The proportion of half rate call arrivals is conversely given by

$$P_{Half} = \frac{\lambda_2}{\lambda_1 + \lambda_2} \qquad \text{... (7.2)}$$

The ***offered traffic per channel*** is given by

$$\rho = \frac{\lambda_1 + \frac{1}{2}\lambda_2}{N\_Channels * \mu} \qquad \text{... (7.3)}$$

where *N_Channels* is the total number of slots available to the users (in this subsection it is eight). The offered traffic per channel should not be unequivocally treated as utilisation, which is defined by [KLEI75] as the 'ratio of the rate at which work enters the system to the maximum rate (capacity) at which the system can perform this work'. Had our frame represented a purely lossless system, then (7.4) would be both the utilisation and offered traffic per channel, as for an M/M/∞ queue. However, since the system under consideration has a finite capacity and blocking of calls is possible, the relationship between utilisation, *U*, and offered traffic per channel, ρ is given by

$$U = (1 - P(Blocking)) * \rho \qquad \text{... (7.4)}$$

In all the cases considered in this research work, it can be stated that ***P(Blocking)*** < 0.02, meaning that the offered traffic per channel provides a very good approximation of utilisation. This offered traffic is the control parameter in all of the graphs, and each curve is obtained at both ρ=0.4 and at ρ=0.7 levels. The call holding time ($1/\mu$) was assumed to be the same for both full and half rate calls, and we set it at 3 minutes, as was done in [ZUKE89]. Figures 7.1 through 7.7 were obtained by simulation, and were then compared to the corresponding data points obtained by the numerically solving the equations resulting from the *Q* matrix. As expected, given the very small 95% confidence interval radii (i.e. a CI radius is given by *(95% CI for Value)/2*Value* and for the simulated probabilities of blocking is always less than 1%) [LARS69] there is virtually no discrepancy between the two sets of points, making it unnecessary to plot both.

Figures 7.1, 7.2 and 7.3 show that, for the Random, First Fit and Best Fit allocation methods respectively, the ***P(Blocking)*** for full and half rate calls behaves differently as the traffic mix is changed. Namely, as we go from a proportion of zero to 100% full rate calls, ***P(Blocking)*** decreases for full rate calls, the opposite being true for half rate calls. This is observed because all three methods *do not achieve good packing compression*, mainly as a result of holes being created upon departures of half rate calls (while Random does not even attempt to "pack tightly" upon arrivals, hence it's the worst of the three). From the point of view of full rate calls, the more half rate calls in the traffic mix, the more chance of isolated half rate calls being around and making it impossible for full rate calls to occupy an empty slot. Half rate calls on the other hand can fit into isolated gaps, so their ***P(Blocking)*** only increases when there are more and more full rate calls in the mix (simple resource contention).

The family of Repacking schemes, shown in Figures 7.4 through 7.7, exhibits different behaviour governed by two effects. The RPR and RPHSR schemes, shown in Figures 7.5 and 7.6, are susceptible only to one of these, the *batch arrivals* effect. If half of a slot is considered to be the basic resource unit of a frame, then full rate call arrivals may be viewed as batch arrivals of two units at a time. Consequently, more batch arrivals (more full rate call arrivals) will cause the system to approach the full state more often. This increases blocking probability for both call types as the proportion of full rate calls increases. Figure 7.4 shows the blocking probability curves for the ordinary Repacking scheme. It is evident that for traffic mixes dominated by fulls (>50%) the curves have almost identical shapes as those for the RPR and RPHSR schemes.

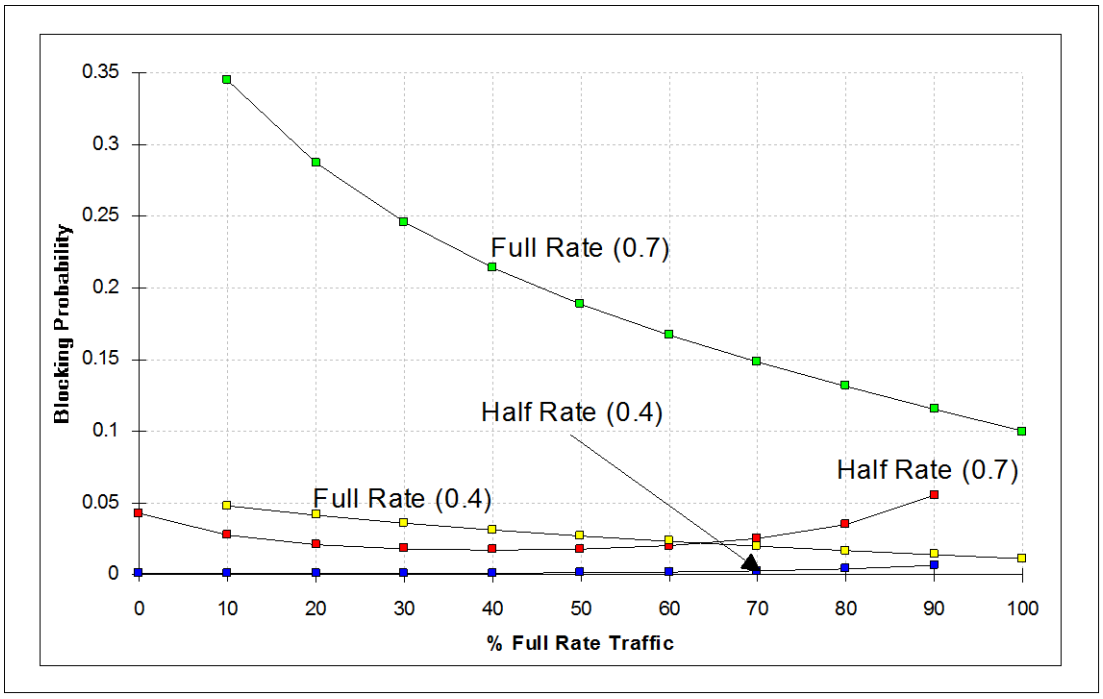

Figure 7.1: Blocking Probability - Random Scheme

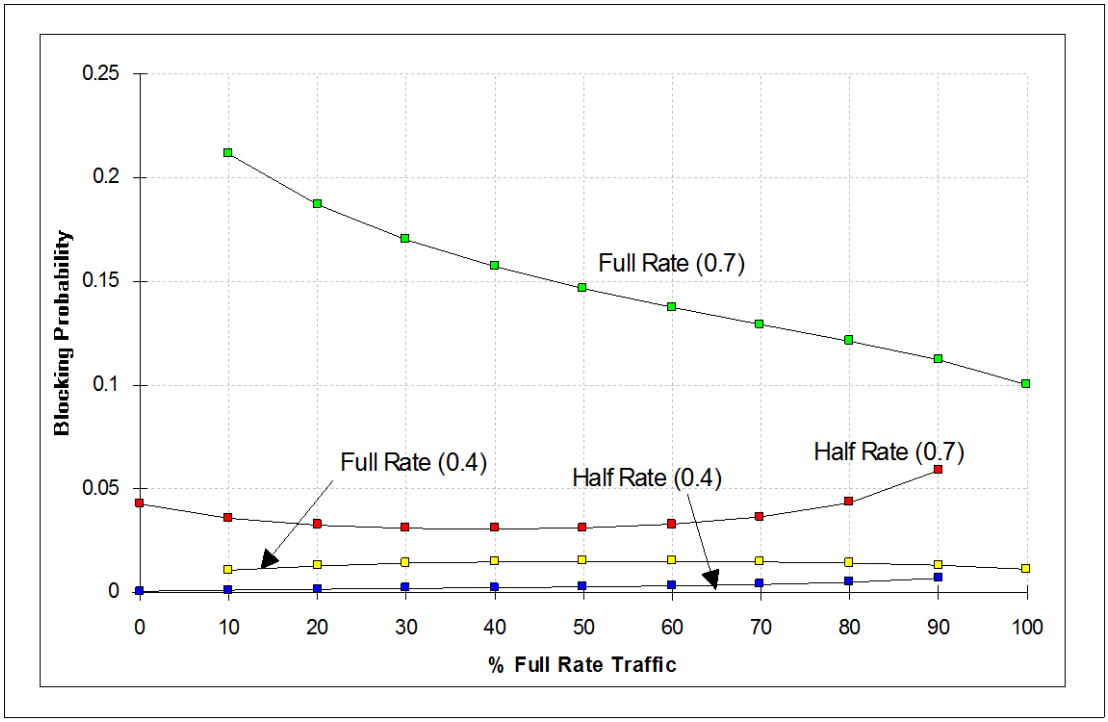

Figure 7.2: Blocking Probability - First Fit Scheme

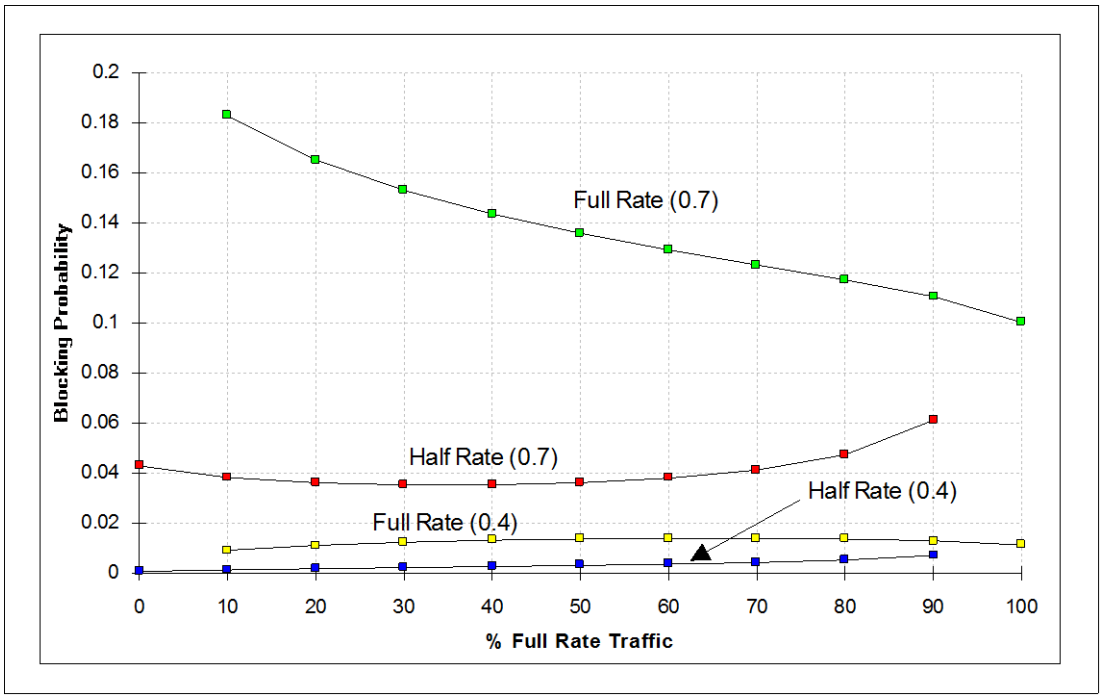

Figure 7.3: Blocking Probability - Best Fit Scheme

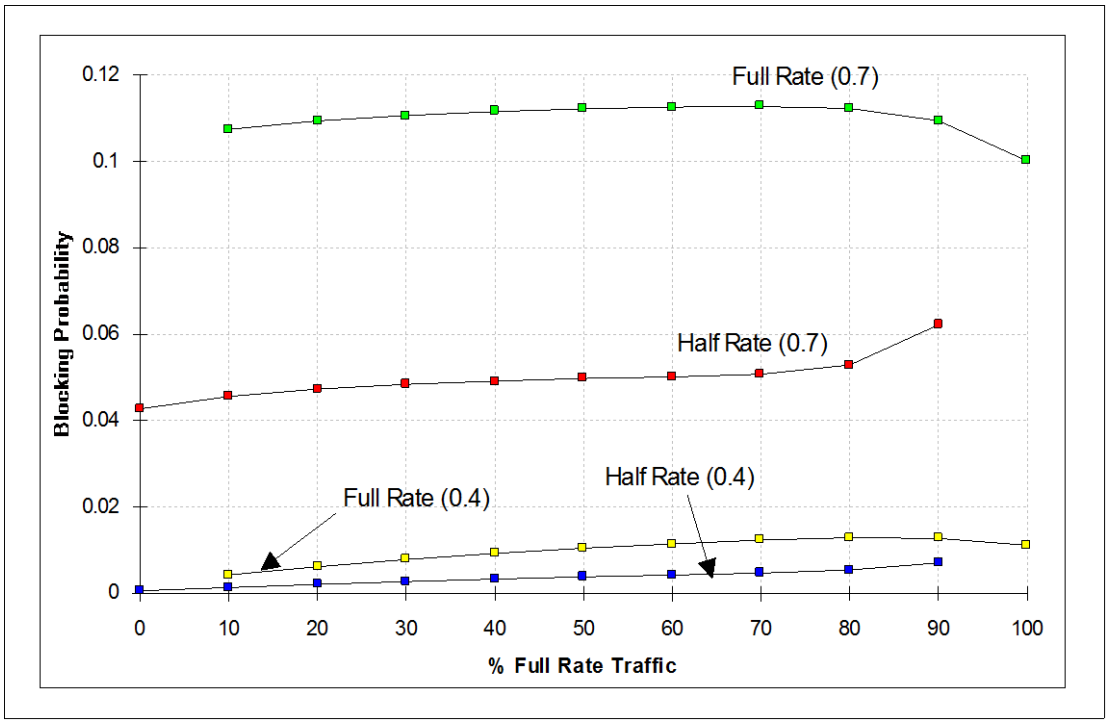

Figure 7.4: Blocking Probability - Repacking Scheme

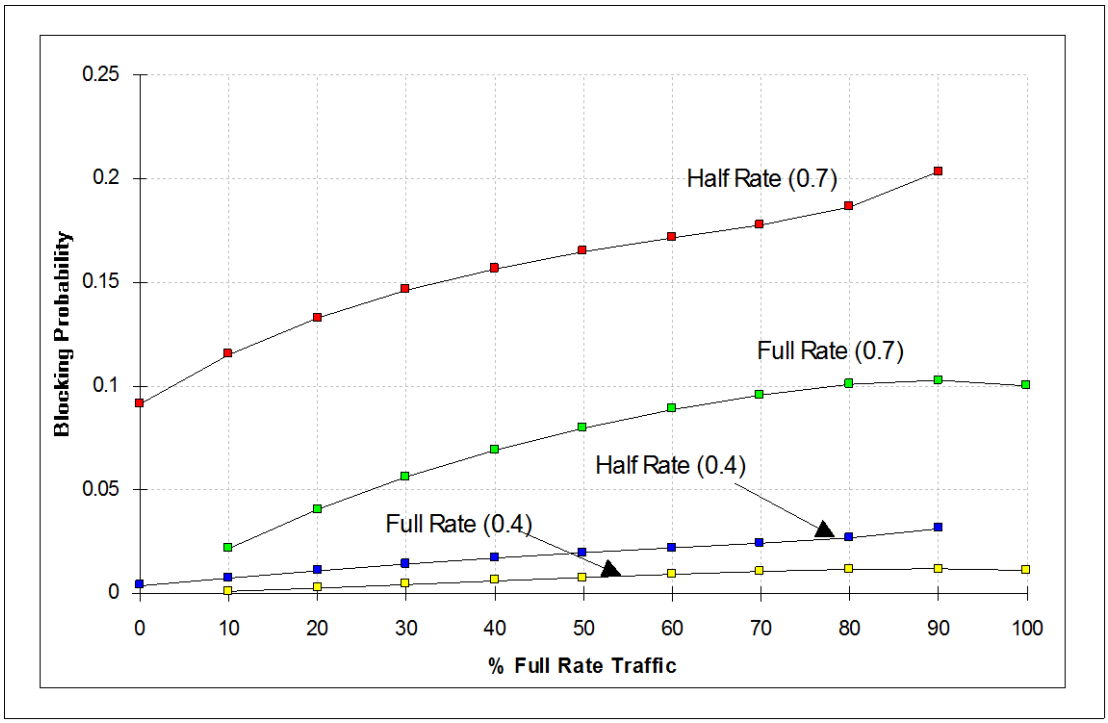

Figure 7.5: Blocking Probability - RPR Scheme

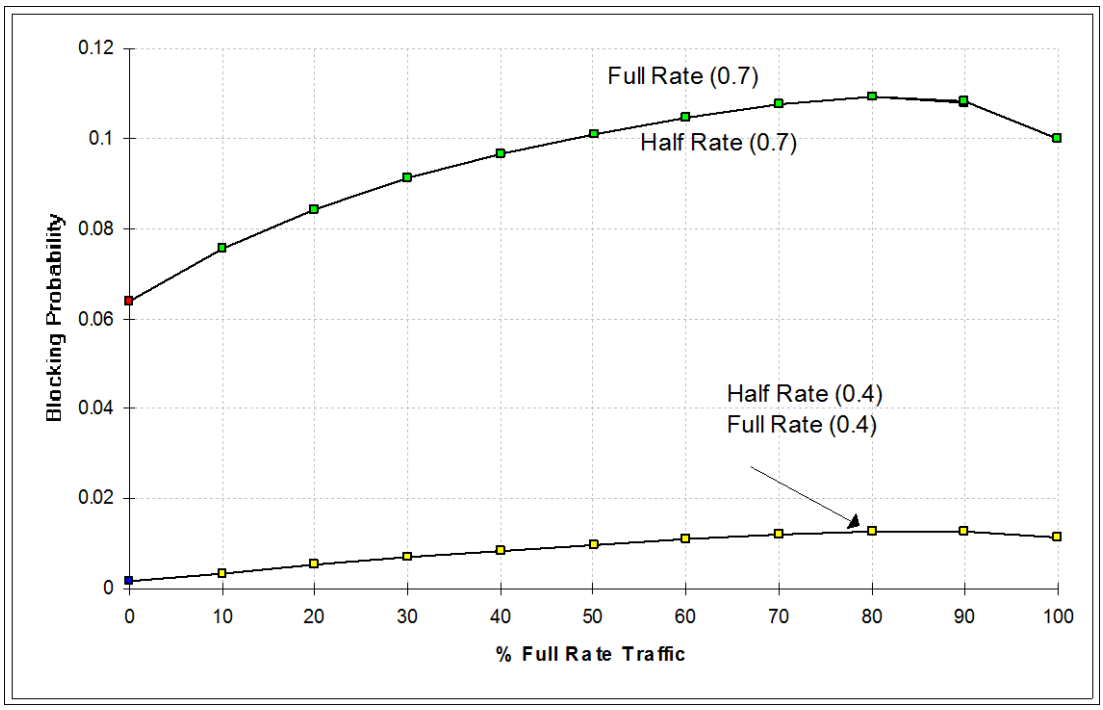

Figure 7.6: Blocking Probability - RPHSR Scheme

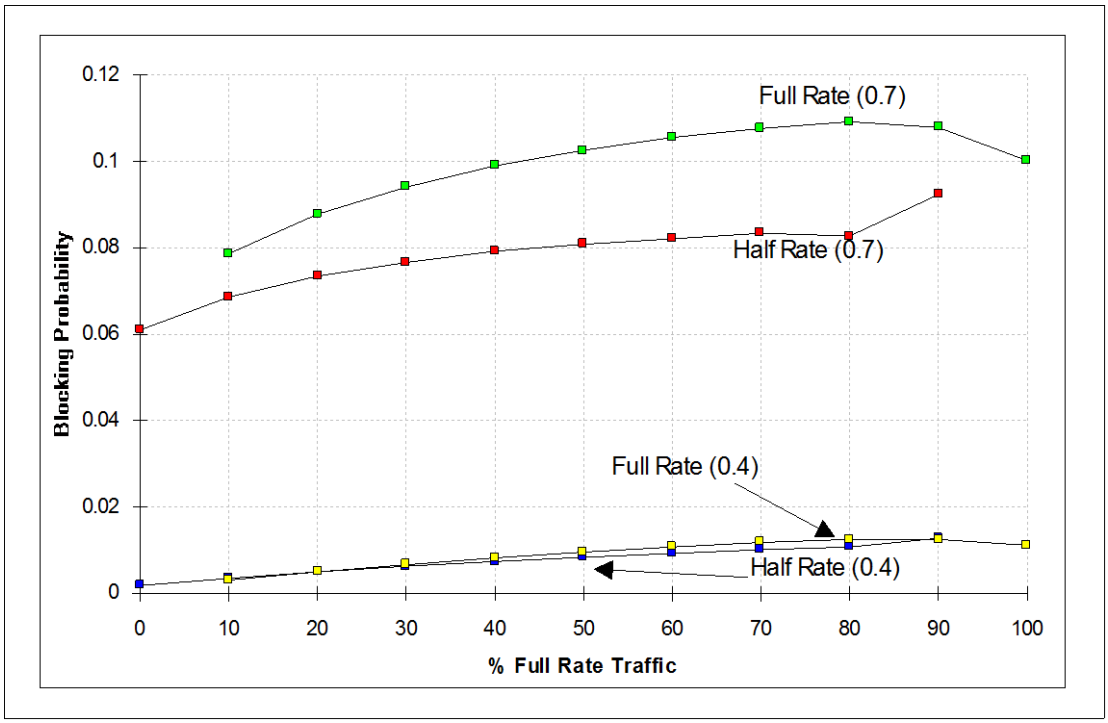

Figure 7.7: Blocking Probability - RRR Scheme

However, when the traffic mix is made up of mostly halves, the loss curves for the two schemes differ in that ordinary Repacking suffers from the *last half slot* effect. Unlike RPR and RPHSR, where the last slot may only be occupied by **one** full exclusively, or by **one** full or **one** half rate call, the Repacking scheme permits two half rate calls to occupy the last available slot. As a result full arrivals experience more loss when there is a high proportion of half rates in the mix. This effect explains why the loss curves for full rate calls in Figure 7.4 are 'flatter' near the origin than the curves in Figures 7.5 and 7.6. Since RRR (p_1=0.6 and p_2 = 1.0) is just a more general form of Repacking (p_1=1.0 and p_2 = 1.0), RPHSR (p_1=1.0 and p_2 = 0) and RPR (p_1=0), Figure 7.7 shows the loss curves for the RRR scheme to lie somewhere in between those of Figures 7.4, 7.5 and 7.6, as it is partially susceptible to both of the above mentioned effects. Figure 7.6 also illustrates that the RPHSR scheme is the only one to provide total fairness (as defined in Section 3, with f = 0) at all levels of offered traffic.

Figures 7.8 and 7.9 illustrate the extreme unfairness inherent in both the Fixed and Sliding Boundary schemes. The common thread in both graphs is that the probability of blocking rises very rapidly (to catastrophic levels) if the traffic mix is made up largely of one type of customer. This is expected, because the scheme is too simplistic in its resource allocation. Namely, either a permanent or dynamically allocated number of slots is reserved for *exclusive use by one type of customers*.

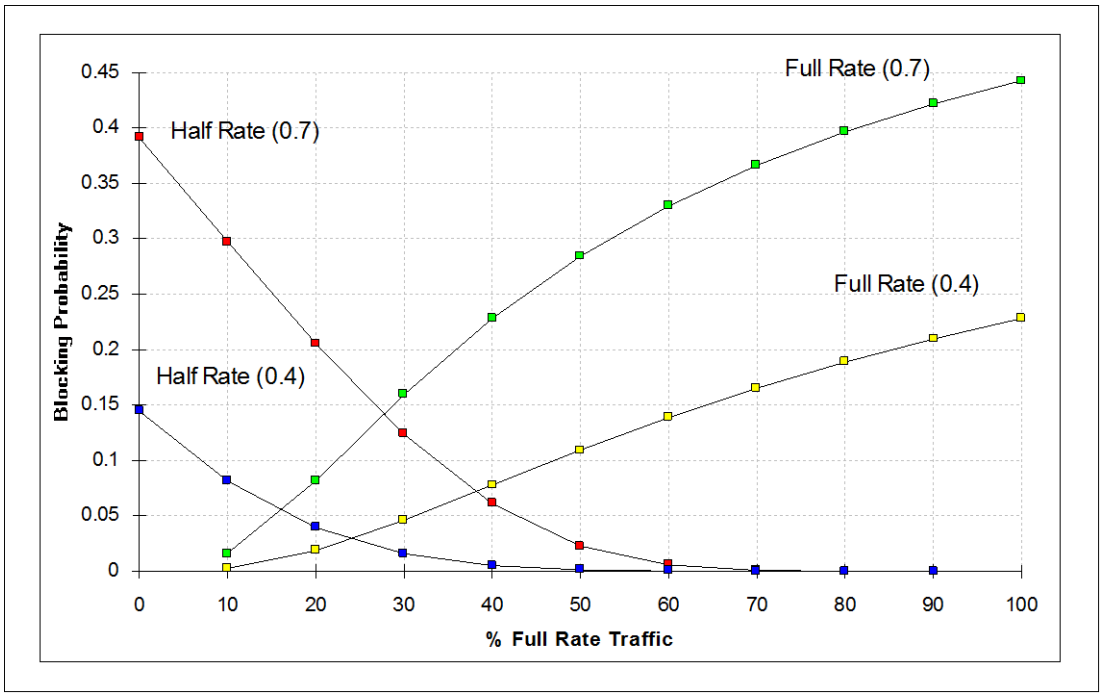

Figure 7.8: Blocking Probability - Fixed Boundary (50/50) Scheme

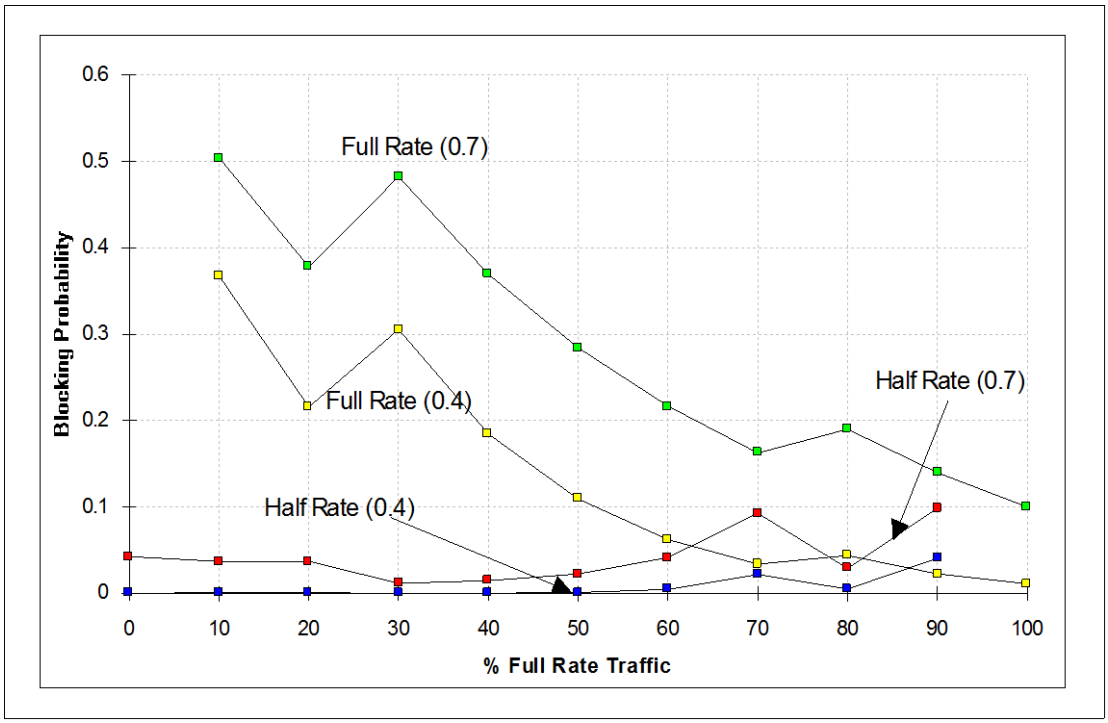

Figure 7.9: Blocking Probability - Sliding Boundary (Traffic Mix Dependent) Scheme

The probability of encountering periods where very few customers of that type arrive is expected to be reasonably high. These periods cause severe wastage of resources and network under-utilisation, and a very likely side-effect of blocking for the other type of customer, who might be experiencing a busy spurt and can never access the free slots, on the other side of the boundary.

Using the definition of fairness in Section 3, Figures 7.10 and 7.11 provide an insight into the logarithm of the ratio of full rate call blocking to half rate call blocking (i.e. the defined *f*-value), at the ρ=0.4 level of offered traffic per channel. The graph in Figure 7.11 shows that the two schemes of mutually-exclusive resource usage (Fixed and Sliding Boundary Schemes) are extremely unfair with $|f| >> 0$, due to the probability of full rate call blocking always being many orders of magnitude larger than its half rate call counterpart. It is for this reason that the Fixed and Sliding Boundary Schemes are judged to be the worst (relative to our other schemes) and are not considered in further analysis, whose aim is to identify the best allocation scheme. Figure 7.10 on the other hand, shows all of the other methods being at least within the same order of magnitude in regard to fairness ($|f| \leq 1$ for all schemes except Random). The reader should note the *convergence towards a mix-insensitive f-value of zero*, as we go from the most unfair scheme, Random allocation, towards the fairest scheme, Repacking with Perpetual Half Slot Reservation.

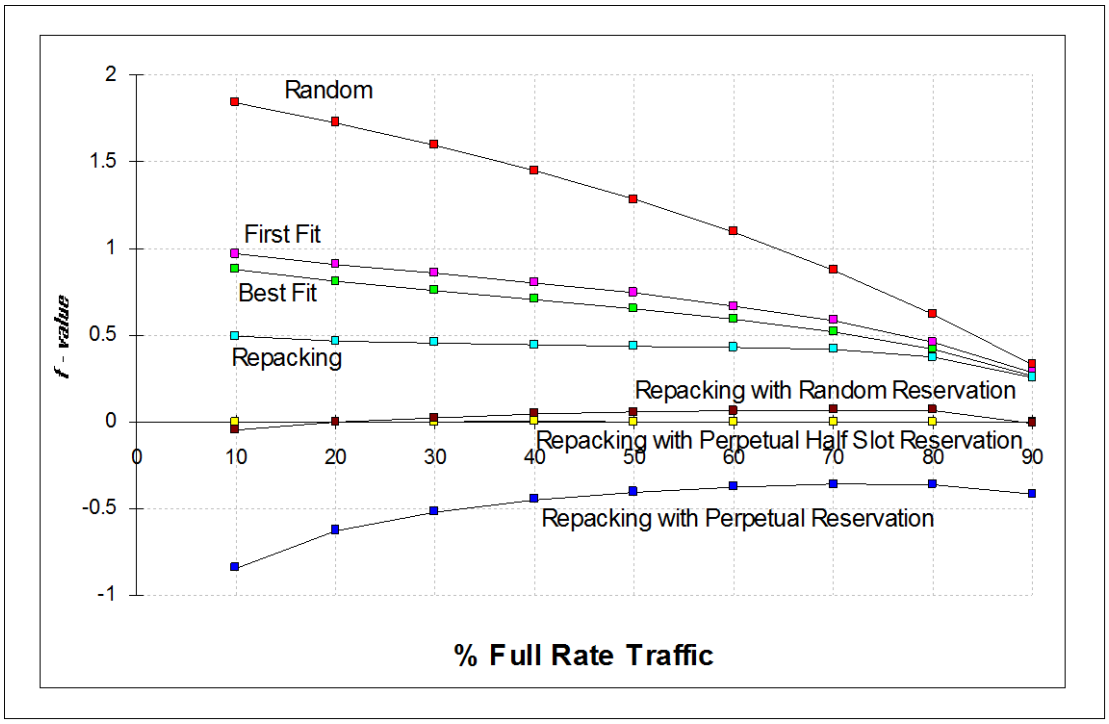

Figure 7.10: Full/Half Rate Call Blocking Fairness - Schemes 3.1 - 3.7 (at ρ=0.4)

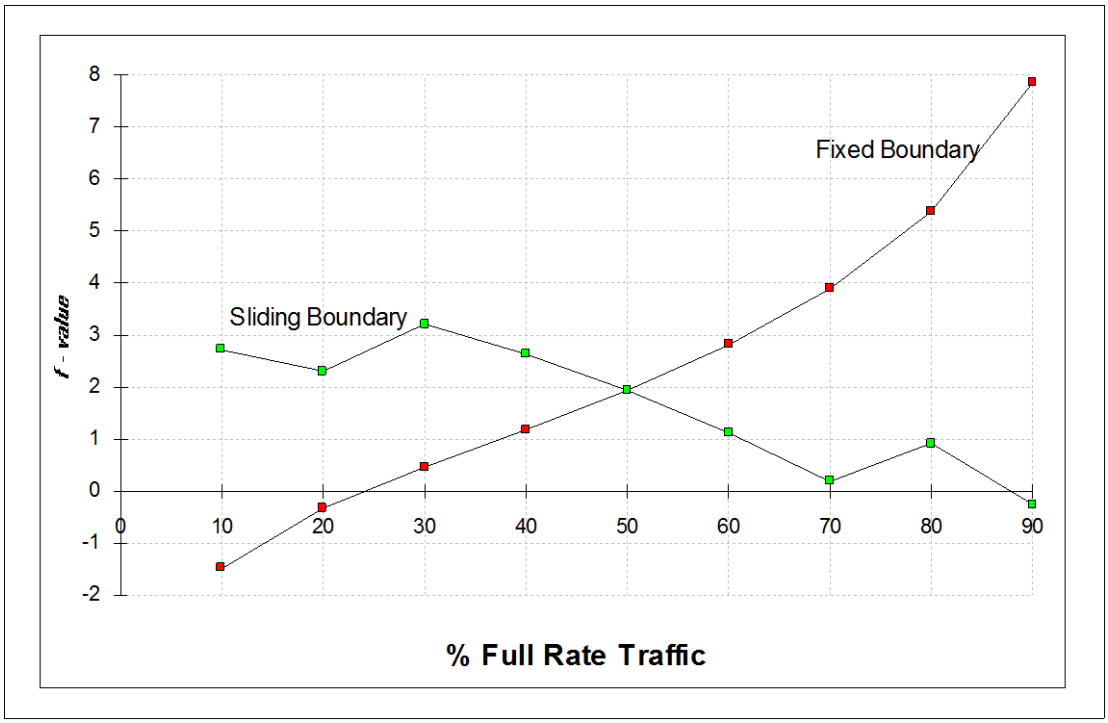

Figure 7.11: Full/Half Rate Call Blocking Fairness - Schemes 3.8 - 3.9 (at ρ=0.4)

The final observation to be made about Fig. 7.10 is that it exemplifies the general statement about access control in Section 3, where it was pointed out that schemes with progressively less access control were progressively more unfair. The Random scheme has no access control with full and half rate calls able to take any slot or hole that is available to them. Such a scheme, as its *f*-value range [1.5, 2] illustrates, is much more unfair than a scheme with stringent access control such as RPHSR, where a range of conditions (i.e. control) is imposed on the access of arriving calls.

7.2 Finding the Most Fair and Efficient Scheme

Having eliminated the Fixed and Sliding Boundary schemes we are left with seven schemes: the Random, First Fit, Best Fit, Repacking, RPR, RPHSR and RRR schemes. Also, *simulation* results in Subsection 7.1 have shown that the blocking probability behaviour and fairness of the First Fit scheme is considerably worse than those of Best Fit, and only slightly better than the worst allocation scheme, Random. Hence it is fair to assume that First Fit can be safely eliminated from the contenders for the best scheme. Figure 7.10 demonstrates clearly that the fairest scheme is RPHSR, in which the algorithm for admission guarantees equality of blocking probability for both types of calls (i.e. $f = 0$ for all traffic mixes). The reason for this observed phenomenon was illustrated by way of Figure 4.6. Namely, each blocking

state for the RPHSR scheme (in an eight slot frame examples are (0,15) or (2,12)) will cause an arriving call to be blocked whether this arriving call is full rate or half rate. This guarantees that the same proportion (not necessarily absolute number) of arriving calls of either type is blocked. Almost equally good in terms of scheme fairness is RRR, which performs only slightly worse. However, RRR is also a more complicated scheme to implement at the physical layer, meaning that the preferred scheme, fairness-wise is RPHSR.

Finding the most efficient scheme is more difficult. In order to accomplish this, it was necessary to find the maximum customer capacity allowed while keeping blocking for both customer types under a certain GOS (a figure of 2% was adopted). This quantity needs to be computed for each scheme, and for a number of traffic mixes ranging from 0% to 100% full rate calls (as a fraction of the total call arrivals), in order to also investigate the sensitivity to traffic-mix. Finding this maximum customer capacity under the GOS constraint required the formulation of an optimisation problem.

The blocking probability *P(Blocking)x* is a function of the proportion of the rate of full rate call arrivals, denoted by θ, and the total system arrival rate, denoted by Λ; where x = 1 for full rate calls; x = 2 for half rate calls. The following relationships are apparent:

$$\theta = \frac{\lambda_1}{\Lambda},$$
$$\Lambda = \lambda_1 + \lambda_2, \qquad \text{... (7.5)}$$
$$C = \Lambda / \alpha.$$

where λ_1, λ_2 are as defined earlier;
C = total number of customers in the system;
α = number of call attempts per unit time per customer.

- Constraints, $\begin{cases} P(Blocking)_1 \le GOS \\ P(Blocking)_2 \le GOS \end{cases}$, where the GOS is chosen to be 2%.

- ***Optimisation Problem: Maximise C, subject to both constraints for any given θ.***

Because C and Λ are related by the positive constant α, which is taken to be identical for all customers, the optimisation problem can now be restated as:

Maximise Λ , subject to both constraints, for any given θ .

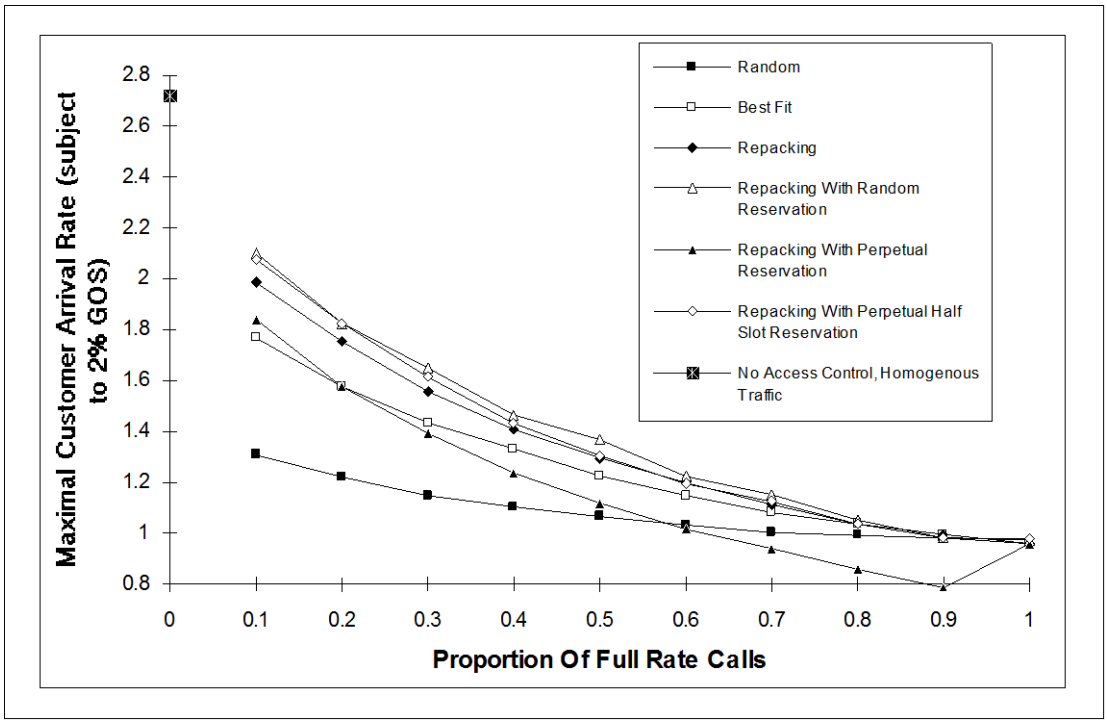

Figure 7.12: One Carrier Frequency System - Comparison of Maximal Customer Arrival Rates subject to a 2% GOS

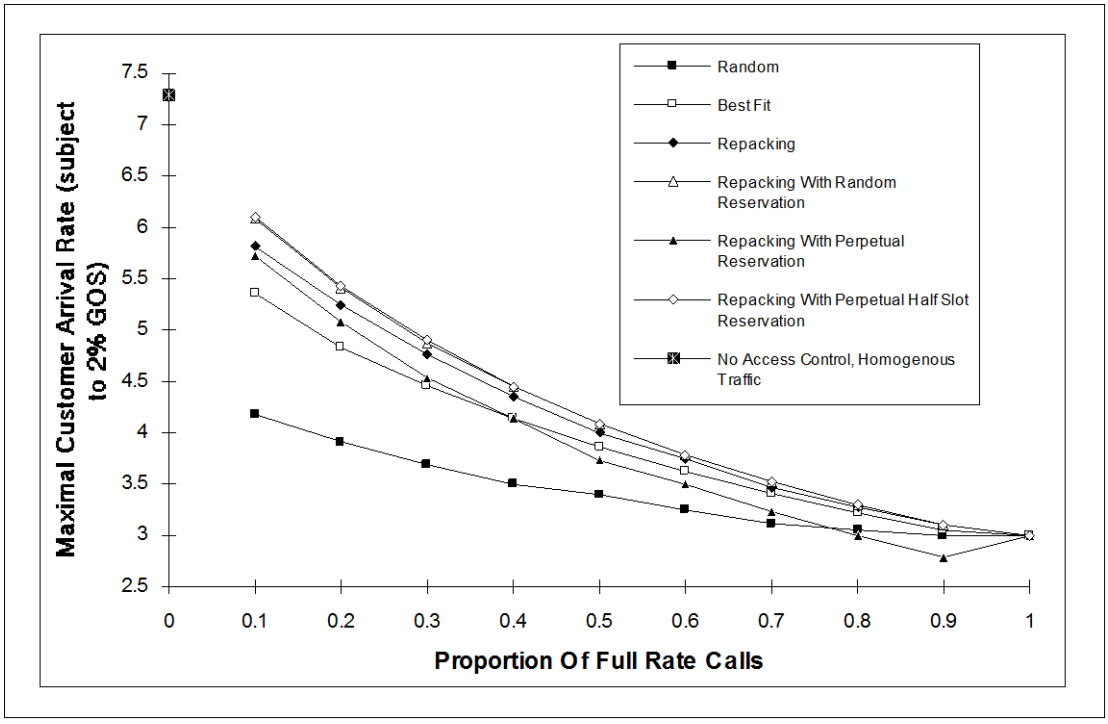

Figure 7.13: Two Carrier Frequency System - Comparison of Maximal Customer Arrival Rates subject to a 2% GOS

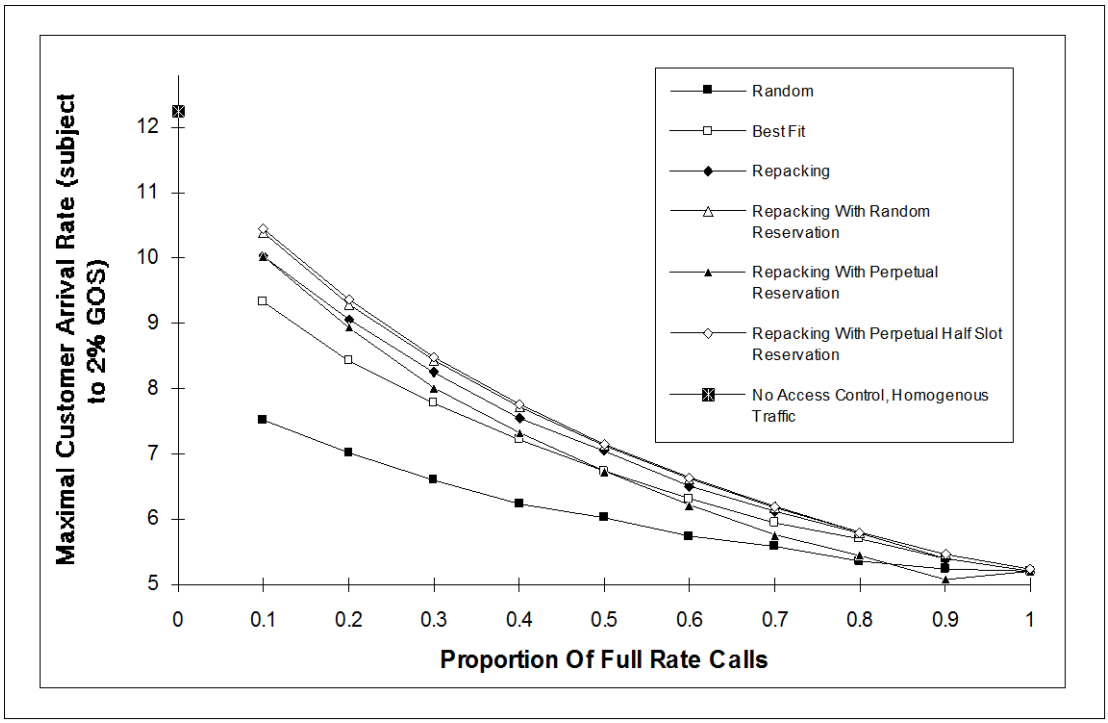

Figure 7.14: Three Carrier Frequency System - Comparison of Maximal Customer Arrival Rates subject to a 2% GOS

Looking at Figures 7.12 through to 7.14, where the maximal value of Λ (the equivalent measure of capacity) is plotted versus θ for systems with one carrier frequency (7 Slots), two carrier frequencies (15 Slots) and three carrier frequencies (23 Slots), it is noticeable that the data point where all the arrivals are half rate calls (i.e. 0% fulls, or $\theta = 0$) is clearly discontinuous from the others. Although Figures 7.12 - 7.14 do not show any points for $\theta < 0.1$, it has been observed that as θ approaches 0:

1. The maximal capacity of the Random, Best Fit and Repacking schemes asymptotically approaches an intermediate value, before making a discontinuous jump to the value shown on the graph at $\theta = 0$ (*ideal peak capacity*). This is due to the fact that at very small values of θ, it is those few present full rate calls which suffer extreme blocking probabilities because of their inability to 'get in'. In order to keep to the pre defined GOS of 2%, the overall user capacity must be constrained. When there are absolutely no full rate calls arriving, this constraint disappears.

2. On the other hand, the maximal capacity of the RRR, RPR, and RPHSR schemes continuously approaches a value which is just below the ideal peak capacity shown on the graph. Because each of these schemes imposes a certain degree of

blocking probability balancing, even when there are very few full rate calls, they are treated approximately equally in terms of access to resources.

The point $\theta = 0$ represents the situation where we are left with **homogenous half rate traffic**. In this case, there is no need to impose any form of access control. The above graphs were produced with the assumption that when this point is reached, none of the schemes is used and the system becomes an M/M/N/N queue and blocking calculated by the Erlang loss formula using N = 2**N_Channels*. In this case the maximum capacity is equal to the ideal peak capacity as shown.

This assumption is particularly well justified in the case of the non reservation schemes, by observing that when $\theta = 0$, the Random, Best Fit and Repacking schemes all yield identical blocking probabilities to those obtained by use of the Erlang-B formula with N = 2**N_Channels.* This holds since none of these schemes prevents access by half rate calls to any part of the frame.

On the other hand, the RPR, RPHSR and RRR schemes all prevent part of the frame resources from being utilised by half rate calls. It was found that, with homogenous half rate traffic this 'prevention of access' causes each of these schemes to give a slightly higher value of blocking probability than that for the Random, Best Fit and Repacking schemes. This happens because in the case of the RPR and RPHSR reservation schemes at $\theta = 0$, N < 2**N_Channels*. In the case of RRR, although it is not an exact Erlang system (i.e. an M/M/N/N queue), its equivalent capacity is less than 2**N_Channels*. With this in mind, in the case of these reservation schemes, it is apparent that the assumption about the system becoming a simple M/M/N/N queue with N = 2**N_Channels* at $\theta = 0$ is somewhat idealised: it is assumed that if the network operator has the knowledge that the network is being used by 100% homogenous half rate traffic, it makes the decision to relax all forms of access control and essentially return to Random slot allocation.

The reality might be that the network operator is unsure that all full rate customers have switched to half rate handsets, and as a result the access control algorithm of the RRR, RPHSR or RPR scheme continues to function (and deny access to some half rate calls) even when there is no need. This explains observation 2. from above, where it is noted that for $\theta = 0$, these three schemes actually yield a maximal capacity slightly lower than the ideal peak capacity shown on the graph.
Given the enormous capacity benefit of having a network with 100% half rate calls, and keeping in mind that there would then be no need for implementing any

allocation scheme other than Random, the network operators should perform economic evaluation studies in order to ascertain whether it would be more profitable to simply upgrade **all** customers to half rate handsets, thereby moving the network to this optimal state.

However, the indicator of scheme performance is obviously the other part of the curves (θ = 0.1 - 1.0 of full rate calls) because given the forecast difficulty in half rate handset market penetration, and the unlikelihood of any operator immediately upgrading **all** customers to half rate handsets, this region will usually be the **real network situation.** With this in mind, it is clear that the optimal performance is again given by the RPHSR and RRR schemes, for all three systems (one carrier frequency, two carrier frequencies and three carrier frequencies). A general feature of all three graphs is that all schemes, in systems of all three sizes, perform increasingly better with a higher proportion of half rate calls. This is intuitively to be expected, because regardless of the scheme, when more half rate calls arrive, we are able to squeeze two users to a single slot more often. Also note that the higher the proportion of half rate calls is, the larger is the benefit gained by employing more complex schemes.

The three figures also highlight that in general, the capacity benefit gained by employing the more efficient repacking family of schemes either reduces with increasing system size, or does not significantly improve. Table 7.1 below compares the capacity benefit of employing the most efficient scheme, RPHSR, instead of a much simpler scheme, Best Fit allocation. Note that the comparison is made for all three system sizes, and at three representative traffic mixes (θ = 0.1, 0.5 and 0.9 of full rate calls).

System size ----->	**One carrier frequency**	**Two carrier frequencies**	**Three carrier frequencies**
Traffic Mix: θ = 0.1	Capacity Benefit =17.5%	Capacity Benefit = 13.9%	Capacity Benefit = 11.9%
Traffic Mix: θ = 0.5	Capacity Benefit =6.5%	Capacity Benefit = 5.9%	Capacity Benefit = 6.1%
Traffic Mix: θ = 0.9	Capacity Benefit =**-1.4%**	Capacity Benefit = 1.7%	Capacity Benefit = 3.1%

Table 7.1: Percentage Capacity Benefit gained by Employing the Most Efficient Scheme (RPHSR) instead of the Simpler to Implement Best Fit Scheme

An interesting feature of the table is the negative capacity benefit observed for a one carrier system with 90% full rate calls and 10% half rate calls (θ = 0.9). This occurs because the RPHSR scheme prevents half rate calls from getting into that final available half slot, while the Best Fit scheme does not. As a result, especially at traffic mixes which have few half rate calls, it is more probable that the half rates will often be attempting to *squeeze* into this last slot with a blocking outcome. Note that the negative value is only observed for the one carrier frequency system, because a reserved half slot represents a wasted 1/14 of total system resources, as opposed to the less significant proportions of 1/30 or 1/66 for the larger systems respectively. Systems of practical significance would certainly have more than one carrier frequency, so from this point of view, the third column in Table 7.1 is the most important one, when making a decision about which scheme to adopt.

The table confirms that this decision cannot be made just once, independently of the traffic mix. Namely, employing a scheme which is more complex to implement than Best Fit, such as RPHSR, does not yield significant capacity improvements in the range where only 10-50% of all customer arrivals are Half Rate (θ = 0.5 to 0.9). An 11.9% capacity improvement is recorded at a time when 90% of all arrivals are Half Rate (θ = 0.1), and this for instance might be considered enough of a benefit to offset the cost of implementing RPHSR over Best Fit. We now summarise the advantages each scheme has over the other:

Best Fit

- Simpler to implement.
- No intracell handover affecting voice quality.

RPHSR

- More efficient in utilising network resources.
- Completely fair in terms of blocking probabilities.

Although the above points are reasonably self explanatory, it should be noted that the most significant downfall of the Best Fit scheme is that it is very unfair, with full rate calls suffering roughly twice the blocking probability of their half rate counterparts. Nonetheless, each of the two schemes has the discussed advantages over the other, and ultimately a dollar value can be assigned to each of these, in order to economically rationalise the cost benefit comparison of implementing one scheme over the other. This is an area for future study.

7.3 The Impact of Intracell Handover on Voice Quality

As discussed in Section 3, each of the Repacking family members would involve the moving of a half rate customer's call from one slot to another, as necessitated by the idea behind the repacking scheme. This moving is termed intracell handover, and involves the customer being placed in a different slot either, (i) within the same carrier frequency frame , or (ii) within a different carrier frequency frame.

Which of the two occurs is irrelevant, since both produce up to a 450 ms period where *voice is cut off* (an audible click is often heard at such times). It is therefore of great interest to the network operator, how badly the Repacking schemes affect the voice quality of half rate calls. We now proceed to investigate the proportion of repackings per carried half rate call for RRR, one of the two best Repacking schemes.

Note that each of the four Repacking schemes will behave approximately identically, as far as intracell handover is concerned, so RRR is a representative sample of the family of schemes (while being equal first in terms of efficiency). The investigation is carried out at the *optimised traffic levels* which were obtained by maximising Λ for the RRR scheme (see Subsection 7.2).

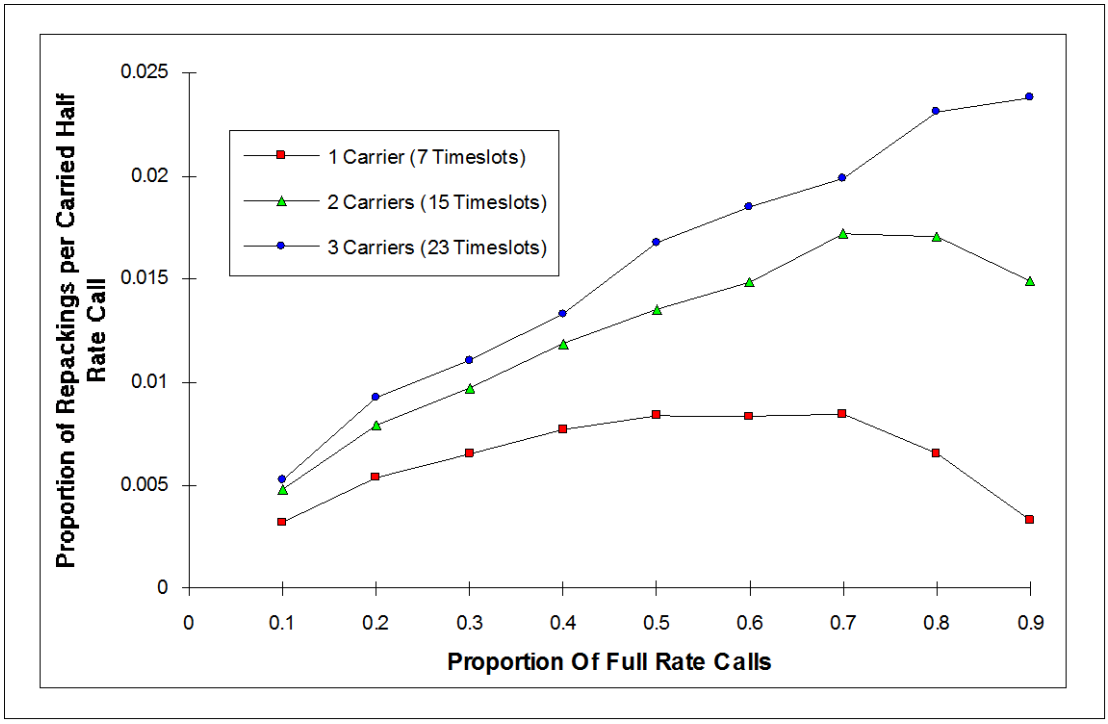

Figure 7.15 Proportion of Repackings per Carried Half Rate Call for the RRR Scheme

The most important observation one can make from Figure 7.15 is that, for all three system sizes, the proportion of carried half rate calls which are repacked is very small, the worst case being 2.4%. As expected, the more half rate calls in the traffic mix, the lower the probability of repack-prompting full rate call arrivals, and this is evident in the curves for all three system sizes. On the other hand, the proportion of repackings per carried half rate call, after an initial period of monotonic increase with increasing θ, begins to again decline as θ approaches 1.0. This can be accounted for by the fact that for θ close to 1.0, there is a smaller number of half rate calls in the system, and hence the probability that full rate calls will arrive at an instant when the frame has enough holes embedded in it to prompt a repacking, is significantly decreased.

Interestingly, the more carrier frequencies in a system, the higher the proportion of repacked half rate calls. This is explained by the fact that *larger systems*, with more carriers, can accommodate a higher offered traffic per channel, for a given GOS. There are three possible events which may take place upon a full rate call arrival: (1) there is an empty slot available and it is allocated with no further action, (2) an empty slot is not available but can be made available by repacking (intracell handover), and (3) an empty slot is not available and can not be made available by repacking. With a higher offered traffic per channel, the probability of Event (1) occurring upon a full rate call arrival is lower.

Given that the GOS is maintained and given that in this analysis the system is optimised to operate at the GOS, the probability of Event (3) occurring upon a full rate call arrival is always bounded by the almost equalised probabilities of blocking for the two traffic types (i.e. the GOS value of 2%). This is true because $P(Blocking)_1 \approx P(Blocking)_2$ for the RRR scheme at most traffic mixes. Hence the probability of Event (2) increases and with it the number of intracell handovers, explaining the higher proportion of half rate calls repacked in larger systems. Even with this in mind, the conclusion which we can draw is that the Repacking family of schemes, which are the most efficient, are almost *negligibly affected by intracell handover-related voice quality degradation.*

7.4 Simplicity, Efficiency and Fairness Comparison

Table 7.2 provides a comprehensive comparison of all schemes, by focussing on the three main criteria of simplicity, efficiency and fairness. Each scheme is awarded a

score out of ten, which is then summed in the last column, giving an overall score. If, for example, RPHSR is the most efficient scheme, its score will be 10/10 for efficiency, with all the other scores given relative to this mark. Note that the row of scores for the First Fit scheme is based on an assumption that its efficiency performance is somewhere in between that of the Random and Best Fit schemes.

Scheme	Implementation Simplicity	Fairness	Efficiency	**Total**
Random	8	3	3	14
First Fit	9	4	5	18
Best Fit	**8**	**7**	**7**	**22**
Repacking	5	7	9	21
RPR	4	8	7	19
RPHSR	**4**	**10**	**10**	**24**
RRR	3	9	10	22
Fixed Boundary	10	0	0	10
Sliding Boundary	9	0	0	9

Table 7.2: Scheme Comparison based on the Simplicity, Efficiency and Fairness Criteria

It is immediately obvious that the RPHSR, RRR and Best Fit are the three best-scoring schemes, in terms of the total. However, implementation simplicity is a significant factor in any scheme consideration, and both Repacking schemes have low scores in this respect, especially RRR (which is the most complex scheme to implement, effectively ruling it out). It was therefore decided to single out a scheme which scored well for simplicity, but also had a good overall score. The only such scheme is Best Fit, and its row, along with the row for RPHSR has been shaded, so as to denote that these two schemes are the ones most worthy of considering for possible implementation in a real network.

Although the three components of the total score are weighted equally in the above scheme comparison, there are cases for either fairness or simplicity to dominate the other criteria. Namely, one could argue that fairness would have to be the most important criterion since customers do not want to pay more for full rate handsets which have better transmission quality, yet experience call blocking several orders of magnitude worse than their cheaper half rate counterparts. On the other hand, it can be argued that implementation simplicity is the limiting factor in any decision by

network operators about which schemes to implement. Schemes which are complex are expensive both in the material sense of implementing them in the necessary hardware, and also in the sense that they consume valuable processing time at either a Mobile Switching Centre (MSC) or at a Base Station Site (BSS) responsible for the call.

The decision on which criteria for comparing the nine schemes should be weighted more than others ultimately lies with the network operator in conjunction with the relevant telecommunications regulatory body (e.g. Austel in Australia).

8. CONCLUSIONS

This thesis has proposed a model for GSM resource management and considered nine channel allocation schemes, with two types of traffic loading: full and half rate calls. The performance of each scheme was determined, based upon the criteria of efficiency, blocking probability equality for both types of user, as well as implementation efficiency. Analytic numerical methods were used to investigate each scheme's efficiency and blocking probability behaviour, and this was successfully compared with simulation results.

Initially a preliminary study of blocking probability and efficiency was carried out for each scheme. The framing structure adopted for this purpose was a generic eight slot frame, without regard to multiple carrier frequencies or any reserved broadcast channels. This study immediately eliminated the Fixed and Sliding boundary schemes from further consideration, due to their extreme unfairness and inefficiency.

The idea behind the eight slot frame was then extended to a more realistic model of GSM, by modelling n=2 and n=3 carrier frequency systems, made up of a reserved control/signalling slot and $8n$-1 user slots. At the expense of significantly increased CPU resources (time, memory) it was still feasible to analytically evaluate the blocking probability and efficiency of six of the nine schemes, (with the exceptions being First Fit and the two already eliminated Fixed and Sliding Boundary schemes).

The results showed that all schemes considered, regardless of system size, perform increasingly better with a higher proportion of half rate calls. It was also found that the higher the proportion of half rate calls was, the larger was the capacity benefit gained by employing more complex schemes. However, the capacity benefit gained by employing the more efficient Repacking family of schemes was found to either reduce with increasing system size, or to not significantly improve.

An overall comparison of the schemes taking into account all traffic mixes was performed with special weight of importance paid to the system with three carrier frequencies (as real network applications will certainly operate on multi-carrier schemes). The comparison was done by way of awarding scores for the three above mentioned criteria and obtaining a total.

The scheme with the highest overall total score was that of Repacking with Perpetual Half Slot Reservation (RPHSR) as it achieved completely equal blocking probabilities for half and full rate calls over a wide range of traffic mixes (total fairness), and it achieved the best maximum customer capacity subject to a GOS constraint (best efficiency). However, this scheme was not very simple to implement, and because of this, was challenged by the Best Fit scheme, which ran a very close second in the overall score. The Best Fit scheme was significantly simpler to implement, with only slightly worse efficiency than RPHSR, and its only downfall was that it did not assure equal blocking probability for both user types.

An important final note needs to be made in order to put these various issues into perspective. As intuitively expected, the point where all the arrivals are half rate calls, yields maximal capacity in terms of arrivals, for each scheme. Given the enormous observed capacity benefit of having a network with 100% half rate customers (see Figures 7.12 - 7.14), and keeping in mind that there would then be no need for implementing any allocation scheme other than Random, the network operators should make it a priority to perform economic evaluation studies in order to ascertain whether it would be more profitable to simply upgrade **all** customers to half rate handsets, thereby moving the network to this optimal state.

9. APPENDIX: C++ CODE LISTINGS

9.1 Simulation Programs

9.1.1 Random Scheme

```
/* Multirate Channel Allocation Simulation: Random Method ___________________________   11/9/95   */

/* Note: This is the simulation, dealing with 8 Full Rate timeslots                              */
/*============================================================================ */

/* Milosh V. Ivanovich , September 1995                                                           */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "rand_por.cc"

#define m1 0.333                /* Avg. 3 min. holding time for both types of calls, as usual in telephony */
#define m2 0.333

/* Prototypes */

void HalfRateArrival(void);            /* The jist of the program --> these perform all the tasks */
void HalfRateDeparture(void);
void FullRateArrival(void);
void FullRateDeparture(void);

void simulate_empty(void);
void simulate_halves(void);
void simulate_fulls(void);
void simulate_all(void);

/* Global Variables */

double l1[11],l2[11];                  /* Arrays of average arrival rate for full and half rate calls */

double HalfBlocked[10][11], FullBlocked[10][11];    /* Arrays for full and half rate call blocking probability */

/* Vars. necessary for calculating the Mean and Confidence Interval size for half and full rate blocking */
double SumHB[11] = {0}, SumFB[11] = {0}, MeanHB[11], S2HB[11], L1HB[11], L2HB[11];
double SumHB2[11] = {0}, SumFB2[11] = {0}, MeanFB[11], S2FB[11], L1FB[11], L2FB[11];

/* Counter vars. for the offered and lost full and half rate calls */
long HalvesAttempted = 0, FullsAttempted = 0, HalvesLost = 0, FullsLost = 0;

/* Counter for the two types of calls presently in system - gives state. k varies proportion of full rate calls */
int Halves = 0, Fulls = 0, IsolHalves = 0, k = 0;

FILE    *fp;       /* File pointer to output data file */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                main
Input Parameters: int argc - Number of command line arguments.
                         char **argv - pointer to command line argument string (the datafile string).
```

```
Output Parameters:          none.
Side-Effects:               Keeps track of system state and depending on current state, calls the appropriate
                  "simulate_xxxx"  functions for different l1[k] and l2[k] values, records blocking
          probability plus confidence interval statistics in output data file (pointed to by
          **argv).
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void main(int argc, char **argv) {

  unsigned long NumEvents, i;      /* Number of events that have occurred so far; i is a multi-purpose counter*/
                                  /* index                                                                  */
  int     seed, run;        /* Arbitrary seed for external RND NUM generator. The run var controls */
                                  /* the no. of times the simulation is performed for Conf. Int. purposes       */
  double   p1;                    /* Proportion of full rate calls in traffic mix.                           */

  /* check command line arguments */
          if (argc != 2) {
            printf("Usage: %s data_file\n", argv[0]);
            exit(1);
          }

   /* check for output data file */
          if ((fp = fopen(argv[1], "w+t")) == NULL) {
            fprintf(stderr, "%s: Can't open %s\n", argv[0], argv[1]);
            exit(1);
          }

 /* printf("\n\n How many events to simulate ??  ");
  scanf("%d", &NumEvents);              */                  NumEvents = 10000000;

 /* printf("\n\n Seed? --> ");
  scanf("%d",&seed);                              */                   seed = 101;
  Myrand.seed(seed);

  for(run=0; run<10; run++)

    for(k=0; k <= 10; k++) {

      HalvesAttempted = FullsAttempted = HalvesLost = FullsLost        /* customary initilization */
      = Halves = Fulls = IsolHalves = 0;

     p1 = k * 0.1;                               /* proportion of FULL RATE traffic                     */
     l1[k] = 0.45*8*m1 * (2*p1/(p1 + 1));        /* arbitrary off. traffic per ch. of 0.45, hence 0.45*8*mu  */
     l2[k] = 2*(0.45*8*m1 - l1[k]);              /* is the average weighted arr. rate                  */

printf("lambda1 = %g; lambda2 = %g ;overall A.R. = l1[k]+0.5*l2[k] = %g\n\n",l1[k],l2[k],(l1[k]+0.5*l2[k]));

     for(i=0; i<NumEvents; i++) {

       if ((Halves==0) && (Fulls==0))                    /* This code calls up the appropriate event    */
        simulate_empty();                                 /* dependent on the given state of the system  */
       else if ((Halves>0) && (Fulls==0))
        simulate_halves();
       else if ((Halves==0) && (Fulls>0))
        simulate_fulls();
       else if ((Halves>0) && (Fulls>0))
        simulate_all();
       else {
```

```
        printf(" This shouldn't happen !!\n");
        exit(1);
      }
    }

  HalfBlocked[run][k] = ((double)HalvesLost / (double)HalvesAttempted);
  SumHB[k] += HalfBlocked[run][k];
  SumHB2[k] += pow(HalfBlocked[run][k],2);

  FullBlocked[run][k] = ((double)FullsLost / (double)FullsAttempted);
  SumFB[k] += FullBlocked[run][k];
  SumFB2[k] += pow(FullBlocked[run][k],2);

 /* printf(" The Probability of Blocking is:  Full Rate Calls       Half Rate Calls ");
  printf("\n -------------------------------------------------------------------");
  printf("\n\t\t\t\t     %lf              %lf           \n\n", FullBlocked[run][k], HalfBlocked[run][k]);
  printf("HAtt = %d , HLost = %d ; FAtt = %d , FLost = %d ;\n\n", HalvesAttempted, HalvesLost,
        FullsAttempted, FullsLost); */
  }

  fprintf(fp, "RANDOM PACKING.   Events = %d.    Seed = %d.\n\n", NumEvents, seed);
  fprintf(fp, "Lambda_tot  ;  Fulls Prop.  ;  HBlock  ;  Left_CI  ;  Right_CI  ;  FBlock  ;  Left_CI  ; ");
  fprintf(fp, " Right_CI\n\n");

  /* Calculate Mean and 95% Confidence Intervals */

  for (k=0; k<=10; k++) {
   MeanHB[k] = SumHB[k] / 10.0;
   S2HB[k] = (10.0/9.0)*((SumHB2[k]/10.0)-pow(MeanHB[k],2));
   L1HB[k] = MeanHB[k] - 2.262*sqrt(S2HB[k])/sqrt(10);
   L2HB[k] = MeanHB[k] + 2.262*sqrt(S2HB[k])/sqrt(10);

   MeanFB[k] = SumFB[k] / 10.0;
   S2FB[k] = (10.0/9.0)*((SumFB2[k]/10.0)-pow(MeanFB[k],2));
   L1FB[k] = MeanFB[k] - 2.262*sqrt(S2FB[k])/sqrt(10);
   L2FB[k] = MeanFB[k] + 2.262*sqrt(S2FB[k])/sqrt(10);

   fprintf(fp, "%5.4lf;  %5.4lf;  %5.4lf;\t %5.4lf;\t %5.4lf;\t %5.4lf;\t %5.4lf;\t %5.4lf\n", (l1[k]+l2[k]), (0.1*k),
MeanHB[k], L1HB[k], L2HB[k], MeanFB[k], L1FB[k], L2FB[k]);

  }
 }

/* Functions*/
/* TYPE OF ALLOCATION : RANDOM                                                    */
/*==========================================================================*/
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                    HalfRateArrival
Input Parameters: none.                      Output Parameters:          none.
Side-Effects:                HalvesAttempted, Halves, IsolHalves and HalvesLost are modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void HalfRateArrival(void) {

    double x, LoneSlots, SharedSlots;

    HalvesAttempted++;                  /* Register the arrival/attempt */
```

```
    if (Halves + 2*Fulls < 16) {

      LoneSlots = double(16 - (2*Fulls + 2*IsolHalves + (Halves-IsolHalves)));
      SharedSlots = double(IsolHalves);
      x = Myrand.uniform(0,(LoneSlots+SharedSlots));
      if (x<LoneSlots) {
         IsolHalves++;
         Halves++;
      }
      else {
         IsolHalves--;
         Halves++;
      }
    }
    else
      HalvesLost += 1;                    /* There wasn't a single half-gap anywhere. Loss occurs.   */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                    HalfRateDeparture
Input Parameters: none.                          Output Parameters:           none.
Side-Effects:                Halves and IsolHalves are modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void HalfRateDeparture(void) {

    double x;

    x = Myrand.uniform(0, double(Halves));
    if (x < double(IsolHalves)) {
       Halves--;
       IsolHalves--;
    }
    else {
       Halves--;
       IsolHalves++;
    }
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                    FullRateArrival
Input Parameters: none.                          Output Parameters:           none.
Side-Effects:                FullsAttempted, Fulls, and FullsLost are modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void FullRateArrival(void) {

    FullsAttempted++;                     /* Register the arrival/attempt */
   if ((Halves-IsolHalves) + 2*Fulls + 2*IsolHalves < 16)
      Fulls++;
   else
      FullsLost += 1;                     /* There wasn't a single full-gap anywhere. Loss occurs.   */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                    FullRateDeparture
Input Parameters: none.                          Output Parameters:           none.
Side-Effects:                Fulls is decremented.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void FullRateDeparture(void) {

      Fulls--;                            /* Nothing much else to do */
```

```
}

/* One of two, three or four events may occurr depending on the state the system is in          */
/* An event can either be the arrival of a Full OR Half rate call, or the departure            */
/* of a Full OR Half rate call. For example, state dependence is illustrated by the fact that  */
/* when there are zero calls in the system, no departures can occurr, so we can only invoke    */
/* the "simulate_empty()" procedure. Each invocation of it calls the RN generator with Real Number  */
/* limits [0,(l1[k]+l2[k])], and determine what has happened.                                  */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                simulate_empty
Input Parameters: none.                    Output Parameters:        none.
Side-Effects:     Depending on randomly obtained value of X, FullRateArrival or HalfRateArrival is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_empty(void) {

   double   X;

   X = Myrand.uniform(0,(l1[k]+l2[k]));
   if (X<l1[k])
     FullRateArrival();
   else
     HalfRateArrival();
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                simulate_halves
Input Parameters: none.                    Output Parameters:        none.
Side-Effects:     Depending on randomly obtained value of X, FullRateArrival, HalfRateArrival  or
                  HalfRateDeparture is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_halves(void) {

    double   X;
    X = Myrand.uniform(0,(l1[k]+l2[k]+Halves*m2));
    if (X<l1[k])
      FullRateArrival();
    else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
          HalfRateArrival();
    else
      HalfRateDeparture();
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                simulate_fulls
Input Parameters: none.                    Output Parameters:        none.
Side-Effects:     Depending on randomly obtained value of X, FullRateArrival, FullRateDeparture,
                  or HalfRateArrival is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_fulls(void) {

    double   X;

    X = Myrand.uniform(0,(l1[k]+l2[k]+Fulls*m1));
    if (X<l1[k])
       FullRateArrival();
    else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
          HalfRateArrival();
    else
      FullRateDeparture();
```

```
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:              simulate_all
Input Parameters: none.                    Output Parameters:      none.
Side-Effects:      Depending on randomly obtained value of X, FullRateArrival, FullRateDeparture,
                   HalfRateArrival  or  HalfRateDeparture is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_all(void) {

   double   X;
   X = Myrand.uniform(0,(l1[k]+l2[k]+Fulls*m1+Halves*m2));
   if (X<l1[k])
      FullRateArrival();
   else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
         HalfRateArrival();
   else if ((X>=l1[k]+l2[k])&&(X<l1[k]+l2[k]+Fulls*m1))
     FullRateDeparture();
   else
     HalfRateDeparture();
}
```

9.1.2 First Fit Scheme

```
/* Multirate Channel Allocation Simulation: First Fit Scheme_______________________________ 3/9/95 */

/* Note: This is the simulation dealing with 8 Full Rate timeslots                        */
/* ======================================================================= */

/* Milosh V. Ivanovich , September 1995                                                  */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "rand_port.cc"

#define m1 0.333            /* Avg. 3 min. holding time for both types of calls, as usual in telephony */
#define m2 0.333

/* Prototypes */

void HalfRateArrival(void);          /* The jist of the program --> these perform all the tasks */
void HalfRateDeparture(void);
void FullRateArrival(void);
void FullRateDeparture(void);

void simulate_empty(void);
void simulate_halves(void);
void simulate_fulls(void);
void simulate_all(void);

/* Global Variables */

double l1,l2;            /* The arrival rates, which will be varied ...            */
int State[8] = {0};      /* Array displaying the state of each channel             */
                         /* 0 - No calls; 1 - One half rate call; 2 - Two half  rate calls; 3 - One  full  */
                         /* rate call.                                             */

int AvailHalves[16] = {0};/* Arrays pointing to the locations in the frame where full or half rate */
int AvailFulls[8] = {0};        /* call departures could randomly occur.             */

double HalfBlocked[10][11], FullBlocked[10][11];   /* Arrays for full and half rate call blocking probability */

/* Vars. necessary for calculating the Mean and Confidence Interval size for half and full rate blocking */
double SumHB[11] = {0}, SumFB[11] = {0}, MeanHB[11], S2HB[11], L1HB[11], L2HB[11];
double SumHB2[11] = {0}, SumFB2[11] = {0}, MeanFB[11], S2FB[11], L1FB[11], L2FB[11];

/* Counter vars. for the offered and lost full and half rate calls */
long HalvesAttempted = 0, FullsAttempted = 0, HalvesLost = 0, FullsLost = 0;

/* Counter for the two types of calls presently in system - gives state. k varies proportion of full rate calls */
int Halves = 0, Fulls = 0, k =0;
FILE    *fp;                    /* File pointer to output data file */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                main
Input Parameters: int argc - Number of command line arguments.
                         char **argv - pointer to command line argument string (the datafile string).
```

```
Output Parameters:          none.
Side-Effects:               Keeps track of system state and depending on current state, calls the appropriate
                    "simulate_xxxx"  functions for different l1[k] and l2[k] values, records blocking
            probability plus confidence interval statistics in output data file (pointed to by
            **argv).
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void main(int argc, char **argv) {

 unsigned long NumEvents, i;      /* Number of events that have occurred so far; i is a multi-purpose counter*/
                                  /* index                                                                   */
  int     seed, run;          /* Arbitrary seed for external RND NUM generator. The run var controls */
                                  /* the no. of times the simulation is performed for Conf. Int. purposes      */
  double   p1;                    /* Proportion of full rate calls in traffic mix.                             */

  /* check command line arguments */
            if (argc != 2) {
              printf("Usage: %s data_file\n", argv[0]);
              exit(1);
            }

   /* check for output data file */
            if ((fp = fopen(argv[1], "w+t")) == NULL) {
              fprintf(stderr, "%s: Can't open %s\n", argv[0], argv[1]);
              exit(1);
            }

 /* printf("\n\n How many events to simulate ??  ");
  scanf("%d", &NumEvents); */                              NumEvents = 10000000;

 /* printf("\n\n Seed? --> ");
  scanf("%d",&seed);          */                           seed = 101;
  Myrand.seed(seed);

  for(run=0; run < 10; run++)

    for(k=0; k <= 10; k++) {

     HalvesAttempted = FullsAttempted = HalvesLost = FullsLost       /* customary initilization */
     = Halves = Fulls = 0;

     for(x=0; x < 8; x++) {
       State[x] = 0;
       AvailFulls[x] = 0;
       AvailHalves[x] = 0; AvailHalves[(2*x + 1)] = 0;
    }

    p1 = k * 0.1;                           /* proportion of FULL RATE traffic                */
    l1 = 0.4*8*m1 * (2*p1/(p1 + 1));          /* arbitrary off. traffic per ch. of 0.4, hence 0.4*8*mu  */
    l2 = 2*(0.4*8*m1 - l1);                 /* is the average weighted arr. rate              */

    printf("lambda1 = %g  ;  lambda2 = %g  ;  overall A.R. = l1+0.5*l2 = %g\n\n", l1,l2,(l1+0.5*l2));

    for(i=0; i<NumEvents; i++) {

      if ((Halves==0) && (Fulls==0))                  /* This code calls up the appropriate event    */
       simulate_empty();                              /* dependent on the given state of the system  */
      else if ((Halves>0) && (Fulls==0))
```

```
       simulate_halves();
      else if ((Halves==0) && (Fulls>0))
       simulate_fulls();
      else if ((Halves>0) && (Fulls>0))
       simulate_all();
      else {
       printf(" This shouldn't happen !!\n");
       exit(1);
      }
    }

  HalfBlocked[run][k] = ((double)HalvesLost / (double)HalvesAttempted);
  SumHB[k] += HalfBlocked[run][k];
  SumHB2[k] += pow(HalfBlocked[run][k],2);

  FullBlocked[run][k] = ((double)FullsLost / (double)FullsAttempted);
  SumFB[k] += FullBlocked[run][k];
  SumFB2[k] += pow(FullBlocked[run][k],2);

/*  printf("\n\n The Probability of Blocking is:  Full Rate Calls       Half Rate Calls ");
  printf("\n ------------------------------------------------------------------------");
  printf("\n\t\t\t\t     %lf             %lf          \n\n", FullBlocked[run][k], HalfBlocked[run][k]); */
  }

  fprintf(fp, "FIRST FIT.   Events = %d.    Seed = %d.\n\n", NumEvents, seed);

/* Calculate Mean and 95% Confidence Intervals */

 for (k=0; k<=10; k++) {
  MeanHB[k] = SumHB[k] / 10.0;
  S2HB[k] = (10.0/9.0)*((SumHB2[k]/10.0)-pow(MeanHB[k],2));
  L1HB[k] = MeanHB[k] - 2.262*sqrt(S2HB[k])/sqrt(10);
  L2HB[k] = MeanHB[k] + 2.262*sqrt(S2HB[k])/sqrt(10);

  MeanFB[k] = SumFB[k] / 10.0;
  S2FB[k] = (10.0/9.0)*((SumFB2[k]/10.0)-pow(MeanFB[k],2));
  L1FB[k] = MeanFB[k] - 2.262*sqrt(S2FB[k])/sqrt(10);
  L2FB[k] = MeanFB[k] + 2.262*sqrt(S2FB[k])/sqrt(10);

  fprintf(fp, "%g\t %g\t %g\t %g\t %g   %g\n", MeanHB[k],
           L1HB[k], L2HB[k], MeanFB[k], L1FB[k], L2FB[k]);
 }
}

/* Functions */
/* TYPE OF ALLOCATION : FIRST FIT                                                    */
/*===================================================================================*/
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                HalfRateArrival
Input Parameters: none.                    Output Parameters:          none.
Side-Effects:            HalvesAttempted, State[i],. Halves, and HalvesLost modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void HalfRateArrival(void) {

   int i,NoGap=0;          /* counter and index variables */
   HalvesAttempted++;              /* Register the arrival/attempt */
   for (i=0; i<8; i++) {           /* First, work out which slots are free */
```

```
      if (State[i] < 2) {
        Halves++;
        State[i] += 1;
        break;
      }
      else
        NoGap++;
  }
  if (NoGap == 8)
     HalvesLost += 1;                          /* There wasn't a single half-gap anywhere. Loss occurs.   */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                    HalfRateDeparture
Input Parameters: none.                        Output Parameters:           none.
Side-Effects:                State[i],. and Halves modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void HalfRateDeparture(void) {

    int i,j,RandNum,Selected;                  /* counter and index variables */
    for(i=0,j=0; i<8; i++) {           /* First, work out which slots are eligible for HR depart.. */
      if (State[i] == 2) {
        AvailHalves[j] = 2*i;
        j++;
        AvailHalves[j] = 2*i + 1;
        j++;
      }
      else if (State[i] == 1) {
        AvailHalves[j] = 2*i;
        j++;
      }
    }

    RandNum = Myrand.uniformInt(0,(j-1));  /* Second, have a "lottery" from which one of the available */
    Selected = AvailHalves[RandNum];        /* half rate channels departs                              */
    State[(Selected/2)] -= 1;
    Halves--;
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                    FullRateArrival
Input Parameters: none.                        Output Parameters:           none.
Side-Effects:                FullsAttempted, State[i],. Fulls, and FullsLost modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void FullRateArrival(void) {

    int i, NoGap = 0;                       /* counter and index variables */
    FullsAttempted++;                          /* Register the arrival/attempt */
    for(i=0; i<8; i++) {                  /* Place in leftmost free FULL slot */
      if (State[i] == 0) {
              Fulls++;
        State[i] = 3;
        break;
      }
      else
        NoGap++;
    }
    if (NoGap == 8)
     FullsLost += 1;                          /* There wasn't a single half-gap anywhere. Loss occurs.   */
```

```
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               FullRateDeparture
Input Parameters: none.                 Output Parameters:      none.
Side-Effects:           State[i] and Fulls modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void FullRateDeparture(void) {

   int i,j,RandNum,Selected;        /* counter and index variables */
   for(i=0,j=0; i<8; i++)           /* First, work out which slots are eligible for FR departure ... */
     if (State[i] == 3) {
       AvailFulls[j] = i;
       j++;
     }
    RandNum = Myrand.uniformInt(0,(j-1)); /* Second, have a "lottery" from which one of the available     */
    Selected = AvailFulls[RandNum];     /* full rate channels departs.                  */
    State[Selected] = 0;
    Fulls--;
}

/* One of two, three or four events may occurr depending on the state the system is in           */
/* An event can either be the arrival of a Full OR Half rate call, or the departure          */
/* of a Full OR Half rate call. For example, state dependence is illustrated by the fact that  */
/* when there are zero calls in the system, no departures can occurr, so we can only invoke        */
/* the "simulate_empty()" procedure. Each invocation of it calls the RN generator with Real Number         */
/* limits [0,(l1+l2)], and determine what has happened.                                 */

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               simulate_empty
Input Parameters: none.                 Output Parameters:      none.
Side-Effects:     Depending on randomly obtained value of X, FullRateArrival or HalfRateArrival is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_empty(void) {

   double   X;
   X = Myrand.uniform(0,(l1+l2));
   if (X<l1)
     FullRateArrival();
   else
     HalfRateArrival();
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               simulate_halves
Input Parameters: none.                 Output Parameters:      none.
Side-Effects:     Depending on randomly obtained value of X, FullRateArrival, HalfRateArrival  or
                  HalfRateDeparture is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_halves(void) {

    double   X;
    X = Myrand.uniform(0,(l1+l2+Halves*m2));
    if (X<l1)
      FullRateArrival();
    else if ((X>=l1)&&(X<l1+l2))
          HalfRateArrival();
    else
      HalfRateDeparture();
```

```
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                simulate_fulls
Input Parameters: none.                    Output Parameters:        none.
Side-Effects:     Depending on randomly obtained value of X, FullRateArrival, FullRateDeparture or
                  HalfRateArrival   is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_fulls(void) {

    double   X;
    X = Myrand.uniform(0,(l1+l2+Fulls*m1));
    if (X<l1)
       FullRateArrival();
    else if ((X>=l1)&&(X<l1+l2))
           HalfRateArrival();
    else
      FullRateDeparture();
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                simulate_all
Input Parameters: none.                    Output Parameters:        none.
Side-Effects:     Depending on randomly obtained value of X, FullRateArrival, FullRateDeparture,
                  HalfRateArrival  or  HalfRateDeparture is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_all(void) {

    double   X;
    X = Myrand.uniform(0,(l1+l2+Fulls*m1+Halves*m2));
    if (X<l1)
       FullRateArrival();
    else if ((X>=l1)&&(X<l1+l2))
           HalfRateArrival();
    else if ((X>=l1+l2)&&(X<l1+l2+Fulls*m1))
      FullRateDeparture();
    else
      HalfRateDeparture();
}
```

9.1.3 Best Fit Scheme

```
/* Multirate Channel Allocation Simulation: Best Fit Scheme _____ 11/9/95                    */

/* Note: This is the simulation dealing with 8 Full Rate timeslots                           */
/*=========================================================================================== */

/* Milosh V. Ivanovich , September 1995                                                      */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "rand_por.cc"

#define m1 0.333                    /* Avg. 3 min. holding time for both types of calls, as usual in telephony */
#define m2 0.333

/* Prototypes */

void HalfRateArrival(void);                 /* The jist of the program --> these perform all the tasks */
void HalfRateDeparture(void);
void FullRateArrival(void);
void FullRateDeparture(void);

void simulate_empty(void);
void simulate_halves(void);
void simulate_fulls(void);
void simulate_all(void);

/* Global Variables */

double l1[11],l2[11];                       /* Arrays of average arrival rate for full and half rate calls */

double HalfBlocked[10][11], FullBlocked[10][11];    /* Arrays for full and half rate call blocking probability */

/* Vars. necessary for calculating the Mean and Confidence Interval size for half and full rate blocking */
double SumHB[11] = {0}, SumFB[11] = {0}, MeanHB[11], S2HB[11], L1HB[11], L2HB[11];
double SumHB2[11] = {0}, SumFB2[11] = {0}, MeanFB[11], S2FB[11], L1FB[11], L2FB[11];

/* Counter vars. for the offered and lost full and half rate calls */
long HalvesAttempted = 0, FullsAttempted = 0, HalvesLost = 0, FullsLost = 0;

/* Counter for the two types of calls presently in system - gives state. k varies proportion of full rate calls */
int k, Halves = 0, Fulls = 0, IsolHalves = 0;

FILE    *fp;        /* File pointer to output data file */

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                   main
Input Parameters: int argc - Number of command line arguments.
                            char **argv - pointer to command line argument string (the datafile string).
Output Parameters:          none.
Side-Effects:               Keeps track of system state and depending on current state, calls the appropriate
                    "simulate_xxxx"  functions for different l1[k] and l2[k] values, records blocking
```

probability plus confidence interval statistics in output data file (pointed to by **argv).

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void main(int argc, char **argv) {

unsigned long NumEvents, i;      /* Number of events that have occurred so far; i is a multi-purpose counter*/
                                 /* index                                                                   */
  int     seed, run;       /* Arbitrary seed for external RND NUM generator. The run var controls */
                                 /* the no. of times the simulation is performed for Conf. Int. purposes     */
  double   p1;                   /* Proportion of full rate calls in traffic mix.                            */

  /* check command line arguments */
          if (argc != 2) {
            printf("Usage: %s data_file\n", argv[0]);
            exit(1);
          }

   /* check for output data file */
          if ((fp = fopen(argv[1], "w+t")) == NULL) {
            fprintf(stderr, "%s: Can't open %s\n", argv[0], argv[1]);
            exit(1);
          }

  /*printf("\n\n How many events to simulate ??  ");
  scanf("%d", &NumEvents);              */      NumEvents = 10000000;

 /* printf("\n\n Seed? --> ");
  scanf("%d",&seed);                          */    seed = 101;
  Myrand.seed(seed);

  for(run=0; run<10; run++)

    for(k=0; k <= 10; k++) {

      HalvesAttempted = FullsAttempted = HalvesLost = FullsLost        /* customary initilization */
      = Halves = Fulls = IsolHalves = 0;

      p1 = k * 0.1;                               /* proportion of FULL RATE traffic              */
      l1[k] = 0.45*8*m1 * (2*p1/(p1 + 1));     /* arbitrary off. traffic per ch. of 0.45, hence 0.45*8*mu  */
      l2[k] = 2*(0.45*8*m1 - l1[k]);             /* is the average weighted arr. rate              */

printf("lambda1 = %g  ;  lambda2 = %g  ;  overall A.R. = l1+0.5*l2 = %g\n\n", l1[k],l2[k],(l1[k]+0.5*l2[k]));

      for(i=0; i<NumEvents; i++) {

           if ((Halves==0) && (Fulls==0))                 /* This code calls up the appropriate event    */
            simulate_empty();                              /* dependent on the given state of the system  */
           else if ((Halves>0) && (Fulls==0))
            simulate_halves();
           else if ((Halves==0) && (Fulls>0))
            simulate_fulls();
           else if ((Halves>0) && (Fulls>0))
            simulate_all();
           else {
            printf(" This shouldn't happen !!\n");
            exit(1);
           }
```

```
    }

  HalfBlocked[run][k] = ((double)HalvesLost / (double)HalvesAttempted);
  SumHB[k] += HalfBlocked[run][k];
  SumHB2[k] += pow(HalfBlocked[run][k],2);

  FullBlocked[run][k] = ((double)FullsLost / (double)FullsAttempted);
  SumFB[k] += FullBlocked[run][k];
  SumFB2[k] += pow(FullBlocked[run][k],2);

/*  printf(" The Probability of Blocking is:  Full Rate Calls       Half Rate Calls ");
  printf("\n ----------------------------------------------------------------------");
  printf("\n\t\t\t\t     %lf            %lf          \n\n", FullBlocked[run][k], HalfBlocked[run][k]);
  printf("HAtt = %d , HLost = %d ; FAtt = %d , FLost = %d ;\n\n", HalvesAttempted, HalvesLost,
        FullsAttempted, FullsLost); */
  }

  fprintf(fp, "BEST FIT.   Events = %d.    Seed = %d.\n", NumEvents, seed);
  fprintf(fp, "Lambda_tot  ;  Fulls Prop.  ;  HBlock  ;  Left_CI  ;  Right_CI  ;  FBlock  ;  Left_CI  ; ");
  fprintf(fp, " Right_CI\n\n");

  /* Calculate Mean and 95% Confidence Intervals */

  for (k=0; k<=10; k++) {
   MeanHB[k] = SumHB[k] / 10.0;
   S2HB[k] = (10.0/9.0)*((SumHB2[k]/10.0)-pow(MeanHB[k],2));
   L1HB[k] = MeanHB[k] - 2.262*sqrt(S2HB[k])/sqrt(10);
   L2HB[k] = MeanHB[k] + 2.262*sqrt(S2HB[k])/sqrt(10);

   MeanFB[k] = SumFB[k] / 10.0;
   S2FB[k] = (10.0/9.0)*((SumFB2[k]/10.0)-pow(MeanFB[k],2));
   L1FB[k] = MeanFB[k] - 2.262*sqrt(S2FB[k])/sqrt(10);
   L2FB[k] = MeanFB[k] + 2.262*sqrt(S2FB[k])/sqrt(10);


   fprintf(fp, "%5.4lf;  %5.4lf;  %5.4lf;\t %5.4lf;\t %5.4lf;\t %5.4lf;\t %5.4lf;\t %5.4lf\n", (l1[k]+l2[k]), (0.1*k),
MeanHB[k], L1HB[k], L2HB[k], MeanFB[k], L1FB[k], L2FB[k]);
  }
 }

/* Functions */
/* TYPE OF ALLOCATION : BEST FIT ( Using States)                                                    */
/*=================================================================================*/
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                HalfRateArrival
Input Parameters: none.                          Output Parameters:          none.
Side-Effects:            HalvesAttempted, Halves, IsolHalves and HalvesLost are modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void HalfRateArrival(void) {

   HalvesAttempted++;                /* Register the arrival/attempt */

   if (Halves + 2*Fulls < 16) {

     if (IsolHalves == 0) {
       IsolHalves++;
       Halves++;
```

```
    }
    else {
      IsolHalves--;
      Halves++;
    }
  }
  else
    HalvesLost += 1;                /* There wasn't a single half-gap anywhere. Loss occurs.   */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               HalfRateDeparture
Input Parameters: none.                     Output Parameters:      none.
Side-Effects:           Halves and IsolHalves are modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void HalfRateDeparture(void) {

  double x;
  x = Myrand.uniform(0, double(Halves));
  if (x < double(IsolHalves)) {
    Halves--;
    IsolHalves--;
  }
  else {
    Halves--;
    IsolHalves++;
  }
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               FullRateArrival
Input Parameters: none.                     Output Parameters:      none.
Side-Effects:           FullsAttempted, Fulls, and FullsLost are modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void FullRateArrival(void) {

  FullsAttempted++;                     /* Register the arrival/attempt */
  if ((Halves-IsolHalves) + 2*Fulls + 2*IsolHalves < 16)
    Fulls++;
  else
    FullsLost += 1;                     /* There wasn't a single full-gap anywhere. Loss occurs.   */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               FullRateDeparture
Input Parameters: none.                     Output Parameters:      none.
Side-Effects:           Fulls is decremented.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void FullRateDeparture(void) {

    Fulls--;                            /* Nothing much else to do */
}

/* One of two, three or four events may occurr depending on the state the system is in          */
/* An event can either be the arrival of a Full OR Half rate call, or the departure            */
/* of a Full OR Half rate call. For example, state dependence is illustrated by the fact that  */
/* when there are zero calls in the system, no departures can occurr, so we can only invoke    */
/* the "simulate_empty()" procedure. Each invocation of it calls the RN generator with Real Number  */
/* limits [0,(l1+l2)], and determines what has happened.                                       */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Function: ***simulate_empty***
Input Parameters: none. Output Parameters: none.
Side-Effects: Depending on randomly obtained value of X, FullRateArrival or HalfRateArrival is called.
~~~~~~*/

```
void simulate_empty(void) {

   double   X;

   X = Myrand.uniform(0,(l1[k]+l2[k]));
   if (X<l1[k])
     FullRateArrival();
   else
     HalfRateArrival();
}
```

/*~~~~~~
Function: ***simulate_halves***
Input Parameters: none. Output Parameters: none.
Side-Effects: Depending on randomly obtained value of X, FullRateArrival, HalfRateArrival or HalfRateDeparture is called.
~~~~~~*/

```
void simulate_halves(void) {

    double   X;

    X = Myrand.uniform(0,(l1[k]+l2[k]+Halves*m2));
    if (X<l1[k])
      FullRateArrival();
    else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
          HalfRateArrival();
    else
      HalfRateDeparture();
}
```

/*~~~~~~
Function: ***simulate_fulls***
Input Parameters: none. Output Parameters: none.
Side-Effects: Depending on randomly obtained value of X, FullRateArrival, FullRateDeparture, or HalfRateArrival is called.
~~~~~~*/

```
void simulate_fulls(void) {

    double   X;

    X = Myrand.uniform(0,(l1[k]+l2[k]+Fulls*m1));
    if (X<l1[k])
       FullRateArrival();
    else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
          HalfRateArrival();
    else
      FullRateDeparture();
}
```

/*~~~~~~
Function: ***simulate_all***
Input Parameters: none. Output Parameters: none.
Side-Effects: Depending on randomly obtained value of X, FullRateArrival, FullRateDeparture, HalfRateArrival or HalfRateDeparture is called.
~~~~~~*/

```
void simulate_all(void) {
```

```
    double   X;

    X = Myrand.uniform(0,(l1[k]+l2[k]+Fulls*m1+Halves*m2));
    if (X<l1[k])
       FullRateArrival();
    else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
           HalfRateArrival();
    else if ((X>=l1[k]+l2[k])&&(X<l1[k]+l2[k]+Fulls*m1))
      FullRateDeparture();
    else
      HalfRateDeparture();
}
```

9.1.4 Repacking Scheme

```
/* Multirate Channel Allocation Simulation: Repacking Scheme _________________ 11/9/95                */

/* Note: This is the simulation dealing with 8 Full Rate timeslots                                    */
/* ================================================================================ */

/* Milosh V. Ivanovich , September 1995                                                               */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "rand_por.cc"

#define m1 0.333                    /* Avg. 3 min. holding time for both types of calls, as usual in telephony */
#define m2 0.333

/* Prototypes */

void HalfRateArrival(void);                 /* The jist of the program --> these perform all the tasks */
void HalfRateDeparture(void);
void FullRateArrival(void);
void FullRateDeparture(void);

void simulate_empty(void);
void simulate_halves(void);
void simulate_fulls(void);
void simulate_all(void);

/* Global Variables */

double l1[11],l2[11];                       /* Arrays of average arrival rate for full and half rate calls */

double HalfBlocked[10][11], FullBlocked[10][11];    /* Arrays for full and half rate call blocking probability */

/* Vars. necessary for calculating the Mean and Confidence Interval size for half and full rate blocking */
double SumHB[11] = {0}, SumFB[11] = {0}, MeanHB[11], S2HB[11], L1HB[11], L2HB[11];
double SumHB2[11] = {0}, SumFB2[11] = {0}, MeanFB[11], S2FB[11], L1FB[11], L2FB[11];

/* Counter vars. for the offered and lost full and half rate calls */
long HalvesAttempted = 0, FullsAttempted = 0, HalvesLost = 0, FullsLost = 0;

/* Counter for the two types of calls presently in system - gives state. k varies proportion of full rate calls */
int k, Halves = 0, Fulls = 0;
FILE    *fp;                            /* File pointer to output data file */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                   main
Input Parameters: int argc - Number of command line arguments.
                            char **argv - pointer to command line argument string (the datafile string).
Output Parameters:          none.
Side-Effects:               Keeps track of system state and depending on current state, calls the appropriate
                    "simulate_xxxx"  functions for different l1[k] and l2[k] values, records blocking
          probability plus confidence interval statistics in output data file (pointed to by
          **argv).
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
```

```
void main(int argc, char **argv) {

 unsigned long NumEvents, i;       /* Number of events that have occurred so far; i is a multi-purpose counter*/
                                   /* index                                                                   */
 int      seed, run;         /* Arbitrary seed for external RND NUM generator. The run var controls */
                                   /* the no. of times the simulation is performed for Conf. Int. purposes        */
 double   p1;                      /* Proportion of full rate calls in traffic mix.                           */

 /* check command line arguments */
           if (argc != 2) {
             printf("Usage: %s data_file\n", argv[0]);
             exit(1);
           }

 /* check for output data file */
    if ((fp = fopen(argv[1], "w+t")) == NULL) {
      fprintf(stderr, "%s: Can't open %s\n", argv[0], argv[1]);
           exit(1);
    }

/* printf("\n\n How many events to simulate ??  ");
 scanf("%d", &NumEvents); */                                  NumEvents = 10000000;

/* printf("\n\n Seed? --> ");
 scanf("%d",&seed);         */                                seed = 101;
 Myrand.seed(seed);

 for(run=0; run<10; run++)

  for(k=0; k <= 10; k++) {

    HalvesAttempted = FullsAttempted = HalvesLost = FullsLost        /* customary initilization */
    = Halves = Fulls = 0;

    p1 = k * 0.1;                              /* proportion of FULL RATE traffic             */
    l1[k] = 0.45*8*m1 * (2*p1/(p1 + 1));       /* arbitrary off. traffic per ch. of 0.45, hence 0.45*8*mu  */
    l2[k] = 2*(0.45*8*m1 - l1[k]);             /* is the average weighted arr. rate           */

printf("lambda1 = %g  ;  lambda2 = %g  ;  overall A.R. = l1[k]+0.5*l2[k] = %g\n", l1[k],l2[k],(l1[k]+0.5*l2[k]));
fprintf(fp,"lambda1 = %g  ;  lambda2 = %g  ;  check total A.R. = l1[k]+l2[k] = %g\n\n",l1[k],l2[k],(l1[k]+l2[k]));

  for(i=0; i<NumEvents; i++) {

    if ((Halves==0) && (Fulls==0))             /* This code calls up the appropriate event      */
     simulate_empty();                         /* dependent on the given state of the system  */
    else if ((Halves>0) && (Fulls==0))
     simulate_halves();
    else if ((Halves==0) && (Fulls>0))
     simulate_fulls();
    else if ((Halves>0) && (Fulls>0))
     simulate_all();
    else {
     printf(" This shouldn't happen !!\n");
     exit(1);
    }
  }
```

```
 HalfBlocked[run][k] = ((double)HalvesLost / (double)HalvesAttempted);
 SumHB[k] += HalfBlocked[run][k];
 SumHB2[k] += pow(HalfBlocked[run][k],2);

 FullBlocked[run][k] = ((double)FullsLost / (double)FullsAttempted);
 SumFB[k] += FullBlocked[run][k];
 SumFB2[k] += pow(FullBlocked[run][k],2);

 printf("\n\n The Probability of Blocking is:  Full Rate Calls       Half Rate Calls ");
 printf("\n ------------------------------------------------------------------------");
 printf("\n\t\t\t\t     %lf            %lf          \n", FullBlocked[k], HalfBlocked[k]);
 printf("HAtt = %d , HLost = %d ; FAtt = %d , FLost = %d ;\n\n", HalvesAttempted, HalvesLost,
       FullsAttempted, FullsLost);
 }

 fprintf(fp, "PURE REPACKING.  Events = %d.   Seed = %d.\n", NumEvents, seed);
 fprintf(fp, "Lambda_tot  ;  Fulls Prop.  ;  HBlock  ;  Left_CI  ;  Right_CI  ;  FBlock  ;  Left_CI  ; ");
 fprintf(fp, " Right_CI\n\n");

 /* Calculate Mean and 95% Confidence Intervals */

  for (k=0; k<=10; k++) {
   MeanHB[k] = SumHB[k] / 10.0;
   S2HB[k] = (10.0/9.0)*((SumHB2[k]/10.0)-pow(MeanHB[k],2));
   L1HB[k] = MeanHB[k] - 2.262*sqrt(S2HB[k])/sqrt(10);
   L2HB[k] = MeanHB[k] + 2.262*sqrt(S2HB[k])/sqrt(10);

   MeanFB[k] = SumFB[k] / 10.0;
   S2FB[k] = (10.0/9.0)*((SumFB2[k]/10.0)-pow(MeanFB[k],2));
   L1FB[k] = MeanFB[k] - 2.262*sqrt(S2FB[k])/sqrt(10);
   L2FB[k] = MeanFB[k] + 2.262*sqrt(S2FB[k])/sqrt(10);

   fprintf(fp, "%5.4lf;  %5.4lf;  %5.4lf;\t %5.4lf;\t %5.4lf;\t %5.4lf;\t %5.4lf;\t %5.4lf\n", (l1[k]+l2[k]), (0.1*k),
MeanHB[k], L1HB[k], L2HB[k], MeanFB[k], L1FB[k], L2FB[k]);

 }
}

/* Functions */
/* TYPE OF ALLOCATION : REPACKING                                                                    */
/*=================================================================================*/
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                HalfRateArrival
Input Parameters: none.                    Output Parameters:          none.
Side-Effects:            HalvesAttempted, Halves, and HalvesLost are modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void HalfRateArrival(void) {

   HalvesAttempted++;              /* Register the arrival/attempt */

   if (Halves + 2*Fulls < 16)
        Halves++;
   else
     HalvesLost += 1;              /* There wasn't a single half-gap anywhere. Loss occurs.   */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                HalfRateDeparture
```

```
Input Parameters: none.                          Output Parameters:         none.
Side-Effects:                Halves is decremented.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void HalfRateDeparture(void) {

    Halves--;                          /* Nothing much else to do */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                    FullRateArrival
Input Parameters: none.                          Output Parameters:         none.
Side-Effects:                FullsAttempted, Fulls, and FullsLost are modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void FullRateArrival(void) {

  FullsAttempted++;                  /* Register the arrival/attempt */
  if (Halves + 2*Fulls < 15)
    Fulls++;
  else
    FullsLost += 1;                  /* There wasn't a single full-gap anywhere. Loss occurs.   */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                    FullRateDeparture
Input Parameters: none.                          Output Parameters:         none.
Side-Effects:                Fulls is decremented.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void FullRateDeparture(void) {
      Fulls--;                         /* Nothing much else to do */
}

/* One of two, three or four events may occurr depending on the state the system is in        */
/* An event can either be the arrival of a Full OR Half rate call, or the departure   */
/* of a Full OR Half rate call. For example, state dependence is illustrated by the fact that            */
/* when there are zero calls in the system, no departures can occurr, so we can only invoke         */
/* the "simulate_empty()" procedure. Each invocation of it calls the RN generator with Real Number */
/* limits [0,(l1[k]+l2[k])], and determine what has happened.                                    */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                    simulate_empty
Input Parameters: none.                          Output Parameters:         none.
Side-Effects:        Depending on randomly obtained value of X, FullRateArrival or HalfRateArrival is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_empty(void) {

   double   X;
   X = Myrand.uniform(0,(l1[k]+l2[k]));
   if (X<l1[k])
     FullRateArrival();
   else
     HalfRateArrival();
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                    simulate_halves
Input Parameters: none.                          Output Parameters:         none.
Side-Effects:        Depending on randomly obtained value of X, FullRateArrival, HalfRateArrival  or
                     HalfRateDeparture is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_halves(void) {
```

```
    double   X;
    X = Myrand.uniform(0,(l1[k]+l2[k]+Halves*m2));
    if (X<l1[k])
      FullRateArrival();
    else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
          HalfRateArrival();
    else
      HalfRateDeparture();
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                   simulate_fulls
Input Parameters: none.                  Output Parameters:          none.
Side-Effects:        Depending on randomly obtained value of X, FullRateArrival, FullRateDeparture,
                     HalfRateArrival is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_fulls(void) {

    double   X;
    X = Myrand.uniform(0,(l1[k]+l2[k]+Fulls*m1));
    if (X<l1[k])
       FullRateArrival();
    else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
          HalfRateArrival();
    else
      FullRateDeparture();
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                   simulate_all
Input Parameters: none.                  Output Parameters:          none.
Side-Effects:        Depending on randomly obtained value of X, FullRateArrival, FullRateDeparture,
                     HalfRateArrival  or  HalfRateDeparture is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_all(void) {

    double   X;
    X = Myrand.uniform(0,(l1[k]+l2[k]+Fulls*m1+Halves*m2));
    if (X<l1[k])
       FullRateArrival();
    else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
          HalfRateArrival();
    else if ((X>=l1[k]+l2[k])&&(X<l1[k]+l2[k]+Fulls*m1))
      FullRateDeparture();
    else
      HalfRateDeparture();
}
```

9.1.5 RPR Scheme

```
/* Multirate Channel Allocation Simulation: Repacking with Perpetual Reservation Scheme_ 11/9/95          */

/* Note: This is the simulation,  dealing with 8 Full Rate channel slots                                   */
/* ======================================================================== */

/* Milosh V. Ivanovich , September 1995                                                                    */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "rand_por.cc"

#define m1 0.333                    /* Avg. 3 min. holding time for both types of calls, as usual in telephony */
#define m2 0.333

/* Prototypes */

void HalfRateArrival(void);                    /* The jist of the program --> these perform all the tasks */
void HalfRateDeparture(void);
void FullRateArrival(void);
void FullRateDeparture(void);

void simulate_empty(void);
void simulate_halves(void);
void simulate_fulls(void);
void simulate_all(void);

/* Global Variables */

double l1[11],l2[11];                                  /* The arrival rates, which will be varied ... */

double HalfBlocked[10][11], FullBlocked[10][11];    /* Arrays for full and half rate call blocking probability */

/* Vars. necessary for calculating the Mean and Confidence Interval size for half and full rate blocking */
double SumHB[11] = {0}, SumFB[11] = {0}, MeanHB[11], S2HB[11], L1HB[11], L2HB[11];
double SumHB2[11] = {0}, SumFB2[11] = {0}, MeanFB[11], S2FB[11], L1FB[11], L2FB[11];

/* Counter vars. for the offered and lost full and half rate calls */
long HalvesAttempted = 0, FullsAttempted = 0, HalvesLost = 0, FullsLost = 0;

/* Counter for the two types of calls presently in system - gives state. k varies proportion of full rate calls */
int k, Halves = 0, Fulls = 0;
FILE    *fp;                            /* File pointer to output data file */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                   main
Input Parameters: int argc - Number of command line arguments.
                            char **argv - pointer to command line argument string (the datafile string).
Output Parameters:          none.
Side-Effects:               Keeps track of system state and depending on current state, calls the appropriate
                    "simulate_xxxx"  functions for different l1[k] and l2[k] values, records blocking
          probability plus confidence interval statistics in output data file (pointed to by
          **argv).
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
```

```
void main(int argc, char **argv) {

 unsigned long NumEvents, i;       /* Number of events that have occurred so far; i is a multi-purpose counter*/
                                   /* index                                                                    */
 int      seed, run;         /* Arbitrary seed for external RND NUM generator. The run var controls */
                                   /* the no. of times the simulation is performed for Conf. Int. purposes       */
 double   p1;                      /* Proportion of full rate calls in traffic mix.                            */

 /* check command line arguments */
           if (argc != 2) {
             printf("Usage: %s data_file\n", argv[0]);
             exit(1);
           }

 /* check for output data file */
    if ((fp = fopen(argv[1], "w+t")) == NULL) {
      fprintf(stderr, "%s: Can't open %s\n", argv[0], argv[1]);
           exit(1);
    }

/* printf("\n\n How many events to simulate ??  ");
 scanf("%d", &NumEvents); */                                  NumEvents = 10000000;

/* printf("\n\n Seed? --> ");
 scanf("%d",&seed);         */                                seed = 101;
 Myrand.seed(seed);

 for(run=0;run < 10; run++)

  for(k=0; k <= 10; k++) {

    HalvesAttempted = FullsAttempted = HalvesLost = FullsLost        /* customary initilization */
    = Halves = Fulls = 0;

    p1 = k * 0.1;                          /* proportion of FULL RATE traffic            */
    l1[k] = 0.45*8*m1 * (2*p1/(p1 + 1));   /* arbitrary off. traffic per ch. of 0.45, hence 0.45*8*mu  */
    l2[k] = 2*(0.45*8*m1 - l1[k]);         /* is the average weighted arr. rate          */

printf("lambda1 = %g  ;  lambda2 = %g  ;  overall A.R. = l1[k]+0.5*l2[k] = %g\n", l1[k],l2[k],(l1[k]+0.5*l2[k]));

  for(i=0; i<NumEvents; i++) {

    if ((Halves==0) && (Fulls==0))              /* This code calls up the appropriate event    */
     simulate_empty();                          /* dependent on the given state of the system  */
    else if ((Halves>0) && (Fulls==0))
     simulate_halves();
    else if ((Halves==0) && (Fulls>0))
     simulate_fulls();
    else if ((Halves>0) && (Fulls>0))
     simulate_all();
    else {
     printf(" This shouldn't happen !!\n");
     exit(1);
    }
  }

 HalfBlocked[run][k] = ((double)HalvesLost / (double)HalvesAttempted);
```

```
 SumHB[k] += HalfBlocked[run][k];
 SumHB2[k] += pow(HalfBlocked[run][k],2);

 FullBlocked[run][k] = ((double)FullsLost / (double)FullsAttempted);
 SumFB[k] += FullBlocked[run][k];
 SumFB2[k] += pow(FullBlocked[run][k],2);

printf("\n\n The Probability of Blocking is:  Full Rate Calls       Half Rate Calls ");
 printf("\n ----------------------------------------------------------------------");
 printf("\n\t\t\t\t     %lf            %lf          \n", FullBlocked[run][k], HalfBlocked[run][k]);
 printf("HAtt = %d , HLost = %d ; FAtt = %d , FLost = %d ;\n\n", HalvesAttempted, HalvesLost,
       FullsAttempted, FullsLost);
}

 fprintf(fp, "REPACKING.   Events = %d.    Seed = %d.\n\n", NumEvents, seed);
 fprintf(fp, "Lambda_tot  ;  Fulls Prop.  ;  HBlock  ;  Left_CI  ;  Right_CI  ;  FBlock  ;  Left_CI  ; ");
 fprintf(fp, " Right_CI\n\n");

 /* Calculate Mean and 95% Confidence Intervals */

 for (k=0; k<=10; k++) {
   MeanHB[k] = SumHB[k] / 10.0;
   S2HB[k] = (10.0/9.0)*((SumHB2[k]/10.0)-pow(MeanHB[k],2));
   L1HB[k] = MeanHB[k] - 2.262*sqrt(S2HB[k])/sqrt(10);
   L2HB[k] = MeanHB[k] + 2.262*sqrt(S2HB[k])/sqrt(10);

   MeanFB[k] = SumFB[k] / 10.0;
   S2FB[k] = (10.0/9.0)*((SumFB2[k]/10.0)-pow(MeanFB[k],2));
   L1FB[k] = MeanFB[k] - 2.262*sqrt(S2FB[k])/sqrt(10);
   L2FB[k] = MeanFB[k] + 2.262*sqrt(S2FB[k])/sqrt(10);

  fprintf(fp, "%5.4lf;  %5.4lf;  %5.4lf;\t %5.4lf;\t %5.4lf;\t %5.4lf;\t %5.4lf;\t %5.4lf\n", (l1[k]+l2[k]), (0.1*k),
MeanHB[k], L1HB[k], L2HB[k], MeanFB[k], L1FB[k], L2FB[k]);
  }
 }

/* Functions */
/* TYPE OF ALLOCATION : REPACKING WITH PERPETUAL (FULL-SLOT) RESERVATION          */
/*=========================================================================*/
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               HalfRateArrival
Input Parameters: none.                 Output Parameters:      none.
Side-Effects:           HalvesAttempted, Halves, and HalvesLost are modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void HalfRateArrival(void) {

   HalvesAttempted++;           /* Register the arrival/attempt */

   if (Halves + 2*Fulls < 14)
        Halves++;
  else
    HalvesLost += 1;            /* There wasn't a single half-gap anywhere. Loss occurs.   */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               HalfRateDeparture
Input Parameters: none.                 Output Parameters:      none.
Side-Effects:           Halves is decremented.
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void HalfRateDeparture(void) {

     Halves--;                          /* Nothing much else to do */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                 FullRateArrival
Input Parameters: none.                    Output Parameters:      none.
Side-Effects:             FullsAttempted, Fulls, and FullsLost are modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void FullRateArrival(void) {

   FullsAttempted++;                    /* Register the arrival/attempt */
   if (Halves + 2*Fulls < 15)
      Fulls++;
   else
      FullsLost += 1;                   /* There wasn't a single full-gap anywhere. Loss occurs.   */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                 FullRateDeparture
Input Parameters: none.                    Output Parameters:      none.
Side-Effects:             Fulls are decremented.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void FullRateDeparture(void) {

      Fulls--;                          /* Nothing much else to do */
}

/* One of two, three or four events may occurr depending on the state the system is in          */
/* An event can either be the arrival of a Full OR Half rate call, or the departure           */
/* of a Full OR Half rate call. For example, state dependence is illustrated by the fact that  */
/* when there are zero calls in the system, no departures can occur, so we can only invoke      */
/* the "simulate_empty()" procedure. Each invocation of it calls the RN generator with Real Number        */
/* limits [0,(l1[k]+l2[k])], and determine what has happened.                                   */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                 simulate_empty
Input Parameters: none.                    Output Parameters:      none.
Side-Effects:      Depending on randomly obtained value of X, FullRateArrival or HalfRateArrival is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_empty(void) {

    double   X;

    X = Myrand.uniform(0,(l1[k]+l2[k]));
    if (X<l1[k])
       FullRateArrival();
    else
       HalfRateArrival();
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                 simulate_halves
Input Parameters: none.                    Output Parameters:      none.
Side-Effects:      Depending on randomly obtained value of X, FullRateArrival, HalfRateArrival  or
                   HalfRateDeparture is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_halves(void) {
```

```
    double   X;

    X = Myrand.uniform(0,(l1[k]+l2[k]+Halves*m2));
    if (X<l1[k])
      FullRateArrival();
    else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
            HalfRateArrival();
    else
      HalfRateDeparture();
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               simulate_fulls
Input Parameters:  none.                  Output Parameters:        none.
Side-Effects:      Depending on randomly obtained value of X, FullRateArrival, FullRateDeparture or
                   HalfRateArrival is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_fulls(void) {

    double   X;
    X = Myrand.uniform(0,(l1[k]+l2[k]+Fulls*m1));
    if (X<l1[k])
       FullRateArrival();
    else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
            HalfRateArrival();
    else
      FullRateDeparture();
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               simulate_all
Input Parameters:  none.                  Output Parameters:        none.
Side-Effects:      Depending on randomly obtained value of X, FullRateArrival, FullRateDeparture,
                   HalfRateArrival  or  HalfRateDeparture is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_all(void) {

    double   X;
    X = Myrand.uniform(0,(l1[k]+l2[k]+Fulls*m1+Halves*m2));
    if (X<l1[k])
       FullRateArrival();
    else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
            HalfRateArrival();
    else if ((X>=l1[k]+l2[k])&&(X<l1[k]+l2[k]+Fulls*m1))
      FullRateDeparture();
    else
      HalfRateDeparture();
}
```

9.1.6 RPHSR Scheme

```
/* Multirate Channel Allocation Sim.: Repacking with Perpetual Half Slot Reservation Scheme 11/10/95      */

/* Note: This is the simulation,  dealing with 8 Full Rate channel slots                                   */
/*  ========================================================================== */

/* Milosh V. Ivanovich , September 1995                                                                    */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "rand_por.cc"

#define m1 0.333                    /* Avg. 3 min. holding time for both types of calls, as usual in telephony */
#define m2 0.333

/* Prototypes */

void HalfRateArrival(void);                 /* The jist of the program --> these perform all the tasks */
void HalfRateDeparture(void);
void FullRateArrival(void);
void FullRateDeparture(void);

void simulate_empty(void);
void simulate_halves(void);
void simulate_fulls(void);
void simulate_all(void);

/* Global Variables */

double l1[11],l2[11];                       /* Arrays of average arrival rate for full and half rate calls */

double HalfBlocked[10][11], FullBlocked[10][11];    /* Arrays for full and half rate call blocking probability */

/* Vars. necessary for calculating the Mean and Confidence Interval size for half and full rate blocking */
double SumHB[11] = {0}, SumFB[11] = {0}, MeanHB[11], S2HB[11], L1HB[11], L2HB[11];
double SumHB2[11] = {0}, SumFB2[11] = {0}, MeanFB[11], S2FB[11], L1FB[11], L2FB[11];

/* Counter vars. for the offered and lost full and half rate calls */
long HalvesAttempted = 0, FullsAttempted = 0, HalvesLost = 0, FullsLost = 0;

/* Counter for the two types of calls presently in system - gives state. k varies proportion of full rate calls */
int k, Halves = 0, Fulls = 0;
FILE    *fp;                            /* File pointer to output data file */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                   main
Input Parameters: int argc - Number of command line arguments.
                            char **argv - pointer to command line argument string (the datafile string).
Output Parameters:          none.
Side-Effects:               Keeps track of system state and depending on current state, calls the appropriate
                    "simulate_xxxx"  functions for different l1[k] and l2[k] values, records blocking
          probability plus confidence interval statistics in output data file (pointed to by
          **argv).
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
```

```
void main(int argc, char **argv) {

 unsigned long NumEvents, i;       /* Number of events that have occurred so far; i is a multi-purpose counter*/
                                   /* index                                                                   */
 int      seed, run;         /* Arbitrary seed for external RND NUM generator. The run var controls */
                                   /* the no. of times the simulation is performed for Conf. Int. purposes        */
 double   p1;                      /* Proportion of full rate calls in traffic mix.                           */

 /* check command line arguments */
           if (argc != 2) {
             printf("Usage: %s data_file\n", argv[0]);
             exit(1);
           }

 /* check for output data file */
    if ((fp = fopen(argv[1], "w+t")) == NULL) {
      fprintf(stderr, "%s: Can't open %s\n", argv[0], argv[1]);
           exit(1);
    }

/* printf("\n\n How many events to simulate ??  ");
 scanf("%d", &NumEvents); */                                  NumEvents = 10000000;

/* printf("\n\n Seed? --> ");
 scanf("%d",&seed);         */                                seed = 101;
 Myrand.seed(seed);

 for(run=0;run < 10; run++)

   for(k=0; k <= 10; k++) {

     HalvesAttempted = FullsAttempted = HalvesLost = FullsLost        /* customary initilization */
     = Halves = Fulls = 0;

     p1 = k * 0.1;                                 /* proportion of FULL RATE traffic          */
     l1[k] = 0.40*8*m1 * (2*p1/(p1 + 1));        /* arbitrary off. traffic per ch. of 0.45, hence 0.45*8*mu  */
     l2[k] = 2*(0.40*8*m1 - l1[k]);              /* is the average weighted arr. rate              */

printf("lambda1 = %g  ;  lambda2 = %g  ;  overall A.R. = l1[k]+0.5*l2[k] = %g\n", l1[k],l2[k],(l1[k]+0.5*l2[k]));

   for(i=0; i<NumEvents; i++) {

     if ((Halves==0) && (Fulls==0))                  /* This code calls up the appropriate event    */
      simulate_empty();                              /* dependent on the given state of the system  */
     else if ((Halves>0) && (Fulls==0))
      simulate_halves();
     else if ((Halves==0) && (Fulls>0))
      simulate_fulls();
     else if ((Halves>0) && (Fulls>0))
      simulate_all();
     else {
      printf(" This shouldn't happen !!\n");
      exit(1);
     }
   }

 HalfBlocked[run][k] = ((double)HalvesLost / (double)HalvesAttempted);
```

```
  SumHB[k] += HalfBlocked[run][k];
  SumHB2[k] += pow(HalfBlocked[run][k],2);

  FullBlocked[run][k] = ((double)FullsLost / (double)FullsAttempted);
  SumFB[k] += FullBlocked[run][k];
  SumFB2[k] += pow(FullBlocked[run][k],2);

/*  printf("\n\n The Probability of Blocking is:  Full Rate Calls       Half Rate Calls ");
  printf("\n ------------------------------------------------------------------------");
  printf("\n\t\t\t\t     %lf            %lf          \n", FullBlocked[run][k], HalfBlocked[run][k]);
  printf("HAtt = %d , HLost = %d ; FAtt = %d , FLost = %d ;\n\n", HalvesAttempted, HalvesLost,
        FullsAttempted, FullsLost); */
 }

  fprintf(fp, "REPACKING WITH PERP. HS. RESERVATION. Rho = 0.40 ; Events = %d.    Seed = %d.\n\n",
NumEvents, seed);
  fprintf(fp, "Lambda_tot  ;  Fulls Prop.  ;  HBlock  ;  Left_CI  ;  Right_CI  ;  FBlock  ;  Left_CI  ; ");
  fprintf(fp, " Right_CI\n\n");

  /* Calculate Mean and 95% Confidence Intervals */

  for (k=0; k<=10; k++) {
    MeanHB[k] = SumHB[k] / 10.0;
    S2HB[k] = (10.0/9.0)*((SumHB2[k]/10.0)-pow(MeanHB[k],2));
    L1HB[k] = MeanHB[k] - 2.262*sqrt(S2HB[k])/sqrt(10);
    L2HB[k] = MeanHB[k] + 2.262*sqrt(S2HB[k])/sqrt(10);

    MeanFB[k] = SumFB[k] / 10.0;
    S2FB[k] = (10.0/9.0)*((SumFB2[k]/10.0)-pow(MeanFB[k],2));
    L1FB[k] = MeanFB[k] - 2.262*sqrt(S2FB[k])/sqrt(10);
    L2FB[k] = MeanFB[k] + 2.262*sqrt(S2FB[k])/sqrt(10);

   fprintf(fp, "%5.4lf;  %5.4lf;  %5.4lf;\t %5.4lf;\t %5.4lf;\t %5.4lf;\t %5.4lf;\t %5.4lf\n", (l1[k]+l2[k]), (0.1*k),
MeanHB[k], L1HB[k], L2HB[k], MeanFB[k], L1FB[k], L2FB[k]);
   }
 }

/* Functions */
/* TYPE OF ALLOCATION : REPACKING WITH PERPETUAL (HALF-SLOT) RESERVATION            */
/*=========================================================================*/
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                HalfRateArrival
Input Parameters: none.                          Output Parameters:          none.
Side-Effects:            HalvesAttempted, Halves, and HalvesLost are modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void HalfRateArrival(void) {

   HalvesAttempted++;              /* Register the arrival/attempt */

   if (Halves + 2*Fulls < 15)
        Halves++;
  else
    HalvesLost += 1;               /* There wasn't a single half-gap anywhere. Loss occurs.   */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                HalfRateDeparture
Input Parameters: none.                          Output Parameters:          none.
```

```
Side-Effects:                 Halves is decremented.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void HalfRateDeparture(void) {

   Halves--;                          /* Nothing much else to do */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                     FullRateArrival
Input Parameters: none.                   Output Parameters:       none.
Side-Effects:                 FullsAttempted, Fulls, and FullsLost are modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void FullRateArrival(void) {

  FullsAttempted++;                   /* Register the arrival/attempt */
  if (Halves + 2*Fulls < 15)
    Fulls++;
  else
    FullsLost += 1;                   /* There wasn't a single full-gap anywhere. Loss occurs.   */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                     FullRateDeparture
Input Parameters: none.                   Output Parameters:       none.
Side-Effects:                 Fulls is decremented.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void FullRateDeparture(void) {

    Fulls--;             /* Nothing much else to do */
}

/* One of two, three or four events may occurr depending on the state the system is in          */
/* An event can either be the arrival of a Full OR Half rate call, or the departure        */
/* of a Full OR Half rate call. For example, state dependence is illustrated by the fact that  */
/* when there are zero calls in the system, no departures can occurr, so we can only invoke      */
/* the "simulate_empty()" procedure. Each invocation of it calls the RN generator with Real Number      */
/* limits [0,(l1[k]+l2[k])], and determine what has happened.                               */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                     simulate_empty
Input Parameters: none.                   Output Parameters:       none.
Side-Effects:       Depending on randomly obtained value of X, FullRateArrival or HalfRateArrival is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_empty(void) {

   double   X;
   X = Myrand.uniform(0,(l1[k]+l2[k]));
   if (X<l1[k])
     FullRateArrival();
   else
     HalfRateArrival();
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                     simulate_halves
Input Parameters: none.                   Output Parameters:       none.
Side-Effects:       Depending on randomly obtained value of X, FullRateArrival, HalfRateArrival  or
                    HalfRateDeparture is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_halves(void) {
```

```
    double   X;
    X = Myrand.uniform(0,(l1[k]+l2[k]+Halves*m2));
    if (X<l1[k])
      FullRateArrival();
    else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
           HalfRateArrival();
    else
      HalfRateDeparture();
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                    simulate_fulls
Input Parameters: none.                        Output Parameters:          none.
Side-Effects:     Depending on randomly obtained value of X, FullRateArrival, FullRateDeparture
                  or HalfRateArrival is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_fulls(void) {

    double   X;
    X = Myrand.uniform(0,(l1[k]+l2[k]+Fulls*m1));
    if (X<l1[k])
       FullRateArrival();
    else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
           HalfRateArrival();
    else
      FullRateDeparture();
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                    simulate_all
Input Parameters: none.                        Output Parameters:          none.
Side-Effects:     Depending on randomly obtained value of X, FullRateArrival, FullRateDeparture,
                  HalfRateArrival  or  HalfRateDeparture is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_all(void) {

    double   X;
    X = Myrand.uniform(0,(l1[k]+l2[k]+Fulls*m1+Halves*m2));
    if (X<l1[k])
       FullRateArrival();
    else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
           HalfRateArrival();
    else if ((X>=l1[k]+l2[k])&&(X<l1[k]+l2[k]+Fulls*m1))
      FullRateDeparture();
    else
      HalfRateDeparture();
}
```

9.1.7 RRR Scheme

```
/* Multirate Channel Allocation Simulation: Repacking with RANDOM Reservation Scheme____ 10/9/95    */

/* Note: This is the simulation dealing with 8 Full Rate timeslots                                  */
/*=========================================================================== */

/* Milosh V. Ivanovich , September 1995                                                             */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "rand_por.cc"

#define m1 0.333                /* Avg. 3 min. holding time for both types of calls, as usual in telephony */
#define m2 0.333

/* Prototypes */

void HalfRateArrival(void);              /* The jist of the program --> these perform all the tasks */
void HalfRateDeparture(void);
void FullRateArrival(void);
void FullRateDeparture(void);

void simulate_empty(void);
void simulate_halves(void);
void simulate_fulls(void);
void simulate_all(void);

/* Global Variables */

double l1[11],l2[11];           /* Arrays of average arrival rate for full and half rate calls        */
P = 0.6;                                 /* The empirically determined optimal value of p1 = P = 0.6,     */
                                         /* where p1 is the probability of allowing a half rate arrival when*/
                                         /* we only have one full timeslot empty.                          */

double HalfBlocked[10][11], FullBlocked[10][11];     /* Arrays for full and half rate call blocking probability */

/* Vars. necessary for calculating the Mean and Confidence Interval size for half and full rate blocking */
double SumHB[11] = {0}, SumFB[11] = {0}, MeanHB[11], S2HB[11], L1HB[11], L2HB[11];
double SumHB2[11] = {0}, SumFB2[11] = {0}, MeanFB[11], S2FB[11], L1FB[11], L2FB[11];

/* Counter vars. for the offered and lost full and half rate calls */
long HalvesAttempted = 0, FullsAttempted = 0, HalvesLost = 0, FullsLost = 0;

/* Counter for the two types of calls presently in system - gives state. k varies proportion of full rate calls */
int k, Halves = 0, Fulls = 0;
FILE    *fp;                    /* File pointer to output data file */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               main
Input Parameters: int argc - Number of command line arguments.
                        char **argv - pointer to command line argument string (the datafile string).
Output Parameters:      none.
Side-Effects:           Keeps track of system state and depending on current state, calls the appropriate
                  "simulate_xxxx"  functions for different l1[k] and l2[k] values, records blocking
```

```
            probability plus confidence interval statistics in output data file (pointed to by
            **argv).
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void main(int argc, char **argv) {

  unsigned long NumEvents, i;      /* Number of events that have occurred so far; i is a multi-purpose counter*/
                                    /* index                                                              */
  int     seed, run;       /* Arbitrary seed for external RND NUM generator. The run var controls */
                                    /* the no. of times the simulation is performed for Conf. Int. purposes        */
  double   p1;                      /* Proportion of full rate calls in traffic mix.                       */

  /* check command line arguments */
            if (argc != 2) {
              printf("Usage: %s data_file\n", argv[0]);
              exit(1);
            }

  /* check for output data file */
      if ((fp = fopen(argv[1], "w+t")) == NULL) {
        fprintf(stderr, "%s: Can't open %s\n", argv[0], argv[1]);
            exit(1);
      }

 /* printf("\n\n How many events to simulate ??  ");
  scanf("%d", &NumEvents); */                              NumEvents = 10000000;

 /* printf("\n\n Seed? --> ");
  scanf("%d",&seed);        */                              seed = 101;
  Myrand.seed(seed);

  for(run=0; run<10; run++)

    for(k=0; k <= 10; k++) {

      HalvesAttempted = FullsAttempted = HalvesLost = FullsLost        /* customary initilization */
      = Halves = Fulls = 0;

      p1 = k * 0.1;                              /* proportion of FULL RATE traffic              */
      l1[k] = 0.45*8*m1 * (2*p1/(p1 + 1));       /* arbitrary off. traffic per ch. of 0.45, hence 0.45*8*mu  */
      l2[k] = 2*(0.45*8*m1 - l1[k]);             /* is the average weighted arr. rate                */

printf("lambda1 = %g  ;  lambda2 = %g  ;  overall A.R. = l1[k]+0.5*l2[k] = %g\n", l1[k],l2[k],(l1[k]+0.5*l2[k]));
fprintf(fp,"lambda1 = %g  ;  lambda2 = %g  ;  check total A.R. = l1[k]+l2[k] = %g  P = %g\n\n",
            l1[k],l2[k],(l1[k]+l2[k]), P);

    for(i=0; i<NumEvents; i++) {

       if ((Halves==0) && (Fulls==0))                    /* This code calls up the appropriate event    */
        simulate_empty();                               /* dependent on the given state of the system  */
       else if ((Halves>0) && (Fulls==0))
        simulate_halves();
       else if ((Halves==0) && (Fulls>0))
        simulate_fulls();
       else if ((Halves>0) && (Fulls>0))
        simulate_all();
       else {
        printf(" This shouldn't happen !!\n");
```

```
      exit(1);
    }
   }

 HalfBlocked[run][k] = ((double)HalvesLost / (double)HalvesAttempted);
 SumHB[k] += HalfBlocked[run][k];
 SumHB2[k] += pow(HalfBlocked[run][k],2);

 FullBlocked[run][k] = ((double)FullsLost / (double)FullsAttempted);
 SumFB[k] += FullBlocked[run][k];
 SumFB2[k] += pow(FullBlocked[run][k],2);

 /*  printf("\n\n The Probability of Blocking is:  Full Rate Calls       Half Rate Calls ");
 printf("\n ---------------------------------------------------------------------------");
 printf("\n\t\t\t\t     %lf            %lf          \n", FullBlocked[k], HalfBlocked[k]);
 printf("HAtt = %d , HLost = %d ; FAtt = %d , FLost = %d ;\n\n", HalvesAttempted, HalvesLost,
       FullsAttempted, FullsLost);  */
}

 fprintf(fp, "REPACKING WITH RANDOM RES. Events = %d.    Seed = %d.\n", NumEvents, seed);
 fprintf(fp, "Lambda_tot  ;  Fulls Prop.  ;  HBlock  ;  Left_CI  ;  Right_CI  ;  FBlock  ;  Left_CI  ; ");
 fprintf(fp, " Right_CI\n\n");

 /* Calculate Mean and 95% Confidence Intervals */

  for (k=0; k<=10; k++) {
   MeanHB[k] = SumHB[k] / 10.0;
   S2HB[k] = (10.0/9.0)*((SumHB2[k]/10.0)-pow(MeanHB[k],2));
   L1HB[k] = MeanHB[k] - 2.262*sqrt(S2HB[k])/sqrt(10);
   L2HB[k] = MeanHB[k] + 2.262*sqrt(S2HB[k])/sqrt(10);

   MeanFB[k] = SumFB[k] / 10.0;
   S2FB[k] = (10.0/9.0)*((SumFB2[k]/10.0)-pow(MeanFB[k],2));
   L1FB[k] = MeanFB[k] - 2.262*sqrt(S2FB[k])/sqrt(10);
   L2FB[k] = MeanFB[k] + 2.262*sqrt(S2FB[k])/sqrt(10);

   fprintf(fp, "%5.4lf;  %5.4lf;  %5.4lf;\t %5.4lf;\t %5.4lf;\t %5.4lf;\t %5.4lf;\t %5.4lf\n", (l1[k]+l2[k]), (0.1*k),
MeanHB[k], L1HB[k], L2HB[k], MeanFB[k], L1FB[k], L2FB[k]);
  }
}
/* Functions */
/* TYPE OF ALLOCATION : REPACKING WITH RANDOM RESERVATION  (RRR)                    */
/*=========================================================================*/
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                HalfRateArrival
Input Parameters: none.                    Output Parameters:          none.
Side-Effects:            HalvesAttempted, Halves, and HalvesLost are modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void HalfRateArrival(void) {

   double Y;                       /* The DIE TOSSING variable */
   HalvesAttempted++;              /* Register the arrival/attempt */

   if (Halves + 2*Fulls < 14)
        Halves++;
  else if (Halves + 2*Fulls == 14) {
            Y = Myrand.uniform(0,1.0);
```

```
    if (Y<P)
       Halves++;
    else HalvesLost += 1;
  }
  else if (Halves + 2*Fulls == 15)
            Halves++;
  else
     HalvesLost += 1;                  /* There wasn't a single half-gap anywhere. Loss occurs.   */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               HalfRateDeparture
Input Parameters: none.                         Output Parameters:          none.
Side-Effects:           Halves is decremented.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void HalfRateDeparture(void) {

    Halves--;                          /* Nothing much else to do */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               FullRateArrival
Input Parameters: none.                         Output Parameters:          none.
Side-Effects:           FullsAttempted, Fulls, and FullsLost are modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void FullRateArrival(void) {

  FullsAttempted++;                    /* Register the arrival/attempt */
  if (Halves + 2*Fulls < 15)
    Fulls++;
  else
    FullsLost += 1;                    /* There wasn't a single full-gap anywhere. Loss occurs.   */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               FullRateDeparture
Input Parameters: none.                         Output Parameters:          none.
Side-Effects:           Fulls is decremented.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void FullRateDeparture(void) {

    Fulls--;                           /* Nothing much else to do */
}

/* One of two, three or four events may occurr depending on the state the system is in                */
/* An event can either be the arrival of a Full OR Half rate call, or the departure                  */
/* of a Full OR Half rate call. For example, state dependence is illustrated by the fact that        */
/* when there are zero calls in the system, no departures can occurr, so we can only invoke          */
/* the "simulate_empty()" procedure. Each invocation of it calls the RN generator with Real Number   */
/* limits [0,(l1[k]+l2[k])], and determine what has happened.                                         */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               simulate_empty
Input Parameters: none.                         Output Parameters:          none.
Side-Effects:       Depending on randomly obtained value of X, FullRateArrival or HalfRateArrival is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_empty(void) {

   double   X;
   X = Myrand.uniform(0,(l1[k]+l2[k]));
   if (X<l1[k])
```

```
      FullRateArrival();
    else
      HalfRateArrival();
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               simulate_halves
Input Parameters: none.                 Output Parameters:      none.
Side-Effects:     Depending on randomly obtained value of X, FullRateArrival, HalfRateArrival  or
                  HalfRateDeparture is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_halves(void) {

     double   X;
     X = Myrand.uniform(0,(l1[k]+l2[k]+Halves*m2));
     if (X<l1[k])
       FullRateArrival();
     else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
             HalfRateArrival();
     else
       HalfRateDeparture();
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               simulate_fulls
Input Parameters: none.                 Output Parameters:      none.
Side-Effects:     Depending on randomly obtained value of X, FullRateArrival, FullRateDeparture or
                  HalfRateArrival is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_fulls(void) {

     double   X;
     X = Myrand.uniform(0,(l1[k]+l2[k]+Fulls*m1));
     if (X<l1[k])
        FullRateArrival();
     else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
             HalfRateArrival();
     else
       FullRateDeparture();
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               simulate_all
Input Parameters: none.                 Output Parameters:      none.
Side-Effects:     Depending on randomly obtained value of X, FullRateArrival, FullRateDeparture,
                  HalfRateArrival  or  HalfRateDeparture is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_all(void) {

     double   X;
     X = Myrand.uniform(0,(l1[k]+l2[k]+Fulls*m1+Halves*m2));
     if (X<l1[k])
        FullRateArrival();
     else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
             HalfRateArrival();
     else if ((X>=l1[k]+l2[k])&&(X<l1[k]+l2[k]+Fulls*m1))
       FullRateDeparture();
     else
       HalfRateDeparture();
}
```

9.1.8 Random Number Generating Program and Header File

```
// Source     : rand_por.cc

//Comments:     This is the external Random Number Generator module, used in all of my simulation
                programs.

// Original Author     : Darren Platt                    Modified by:  Milosh Ivanovich

// Version    : 1.1
// Date Begun : 10.5.91
// Last Rev   : 9/9/1995
// Id String  : @(#)rand.cc  1.1
//
// Modification log
//-----------------
// 0.0  tjp                    Tom was here.
// 1.0      daz                Expanded to use a class
// 1.1      daz                Added uniformInt and put in SCCS
// 1.2  mil            removed main() to link with my simulation sim.XX series.
//
//
// Portable random number generator.
// Modified lahmer, generates integers in range min_long to max_long
// 1.6.91 Expanded to include a random number class.

static char *WhatString = "@(#)rand.cc        1.1";

#include "rand.h"
#include <assert.h>
#include <math.h>
#include <iostream.h>

unsigned long      global_table[128];
unsigned long      *rand_table = global_table;
unsigned long      lahmer_seed;
long   taus_seed;
unsigned long   seed;
      double   L,U,mu,stdev;      /* Global Vars */
        int   nums, i;          /* Number of Generated RNs and Universal Counter */

void seed_lahmer(long seed)
{
          lahmer_seed = seed;
}
unsigned long lahmer(void)
{
          return lahmer_seed=(69069l*lahmer_seed+1);
}
void fill_table(void)
{
          for(int i=0;i<128;i++)
                    rand_table[i] = lahmer();
}
void seed_taus(long seed)
{
```

```
        taus_seed = seed;
}
long taus(void)
{
        const long k_value = 123456;
        if (taus_seed >=0) return (taus_seed = (taus_seed<<1)%1000003);
        else return taus_seed = ((taus_seed<<1)^k_value)%1000003;
}
void seed_random(unsigned long seed)
{
        seed_lahmer(seed);
        seed_taus((long)seed);
        fill_table();
}

unsigned long my_rand(void)
{
        int choice = taus()%128;
        if (choice<0) choice = -choice;
        unsigned long ret_value = rand_table[choice];
        rand_table[choice] = lahmer();
        return ret_value;
}
//
// Method definitions for the randcl class
//
inline double randcl::quick_unit(void)
{
        return (my_rand()+0.5)/(double)max_ulong;
}
inline void randcl::ungrab(void)
{
        taus_seed = ::taus_seed;
        lahmer_seed = ::lahmer_seed;
        rand_table = ::rand_table;
}
inline void randcl::grab(void)
{
        ::taus_seed = taus_seed;
        ::lahmer_seed = lahmer_seed;
        ::rand_table = rand_table;
}

void randcl::seed(unsigned long seed)
{
        grab();
        seed_random(seed);
        ungrab();
}
randcl::randcl(void)
{
        assert(rand_table = new unsigned long[128]);
        seed(1);
        phase = 0;
}
randcl::randcl(unsigned long seed)
{
```

```
	assert(rand_table = new unsigned long[128]);
	randcl::seed(seed);
	phase=0;
}
double randcl::neg_exp(double mean)
{
	return -mean*log(unit());
}
double randcl::normal(double mean,double sd)
{
	if (phase) {		// Already have one stored up.
		phase = 0;
		return (sqratio * q * sd)+mean;
	}
	double	p,v;
	grab();
	do {
		p = quick_unit()*2-1; q = quick_unit()*2-1;
		v = p*p + q*q;
	} while(v > 1.0 || v <0.25);
	sqratio = sqrt(-2*log(quick_unit()) / v);
	ungrab();
	phase = 1;
	return (sd * sqratio * p)+mean;
}
long randcl::uniformInt(long lower,long upper)
{
	return (ulong()%(upper-lower+1))+lower;
}

double randcl::uniform(double lower,double upper)
{
	return (upper-lower)*unit()+lower;
}
randcl::~randcl()
{
	delete rand_table;
}
double randcl::unit(void)
{
	grab();
	unsigned long	ul = my_rand();
	ungrab();
	return (ul+0.5)/(double)max_ulong;
}
unsigned long randcl::ulong(void)
{
	grab();
	unsigned long ul = my_rand();
	ungrab();
	return ul;
}
```

Header File

```
// Source     : rand.h

// Comments:    This is the header file for the portable Random Number Generator "rand_por.cc" which
//              was used as an external modeule in every simulation program.

// Original Author     : Darren Platt          Modifiied : Milosh Ivanovich

// Version    : 1.1
// Date Begun : 10.5.91
// Last Rev   : 9/9/1995

// Id String  : @(#)rand.h   1.1
//
// Modification log
//-----------------
// 0.0  tjp                 Tom was here.
// 1.1      daz             Added to SCCS

#ifndef _RANDOM_
#define _RANDOM_
void seed_random(unsigned long);
unsigned long my_rand(void);

const unsigned long max_ulong = 0xffffffff;

// The random number generator class contains a number of methods for
// generating random numbers with different distributions. It also has
// its own seeds making it good for use in simulations.

class randcl {
                unsigned long   *rand_table;
                unsigned long   lahmer_seed;
                int                     phase;
                double                  sqratio,q;
                long                    taus_seed;
                inline  void    grab(void);
                inline  void    ungrab(void);
                inline  double  quick_unit(void);
        public:
                                        randcl(unsigned long);      // seed
                                        randcl(void);
                                        ~randcl();
        unsigned long   ulong(void);
                void            seed(unsigned long);
                double          unit(void);
                double          neg_exp(double);            // mean
                double          uniform(double,double);     // lower,upper
                long            uniformInt(long,long);      // lower,upper
                double          normal(double,double);      // mean,sd
};
#endif
```

9.1.9 Voice Quality Impact Simulation

```
/* Multirate Channel Allocation : Voice Quality Impact Simulation: RRR Scheme _____ 3/9/95        */

/* Note: This is the simulation gauging QOS, dealing with the 7, 15 or 23 Full Rate timeslots      */
/* ================================================================================ */

/* Milosh V. Ivanovich , September 1995                                                          */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "rand_por.cc"

#define m1 0.333                    /* Avg. 3 min. holding time for both types of calls, as usual in telephony */
#define m2 0.333

/* Prototypes */

void HalfRateArrival(void);                 /* The jist of the program --> these perform all the tasks */
void HalfRateDeparture(void);
void FullRateArrival(void);
void FullRateDeparture(void);

void simulate_empty(void);
void simulate_halves(void);
void simulate_fulls(void);
void simulate_all(void);

/* Global Variables */

int  Slots;                         /* Number of  slots available to the users (e.g. 7, 15 or 23) */

/* The arrival rates are specifically chosen to match the traffic conditions yielding peak throughput while
satisfying the < 2% GOS criterium, for this, the RRR scheme.                                  */

double l1[9] = {0.208276, 0.364080, 0.479520, 0.578468, 0.657120, 0.719280, 0.789796, 0.828800,
                0.832850 };

double l2[9] = {1.874484, 1.456320, 1.118880, 0.867702, 0.657120, 0.479520, 0.338484, 0.207200,
                0.092539 };

int State[24] = {0};        /* Array displaying the state of each channel                        */
                            /* 0 - No calls; 1 - One half rate call; 2 - Two half  rate calls; 3 - One full rate call  */

int AvailHalves[48] = {0};          /* Arrays containing pointers to full and half rate calls which are*/
int AvailFulls[24] = {0}, Eligible[24] = {0};     /* available to depart (at random) from their timeslot   */
                                    /* locations, and an array of pointers to locations with half */
                                    /* rate calls eligible for repacking.                         */

/* Arrays for full and half rate call blocking probability, and proportion of repacked half rate calls */
double HalfBlocked[10][9], FullBlocked[10][9], HRep[10][9];

/* Vars. necessary for calculating the Mean and Confidence Interval size for half and full rate blocking */
double SumHB[9] = {0}, SumFB[9] = {0}, MeanHB[9], S2HB[9], Diam_HB[9];
```

```
double SumHB2[9] = {0}, SumFB2[9] = {0}, MeanFB[9], S2FB[9], Diam_FB[9];
double SumHRep[9] = {0}, SumHRep2[9] = {0}, MeanHRep[9], VarHRep[9], Diam_HRep[9];

/* Counter vars. for the offered and lost full and half rate calls, and the repacked half rate calls */
long HalvesAttempted = 0, FullsAttempted = 0, HalvesLost = 0, FullsLost = 0;
long HalvesRepacked = 0;

/* Counter for the two types of calls presently in system - gives state. k varies proportion of full rate calls */
int  prop, k, Halves = 0, Fulls = 0;
FILE    *fp;                          /* File pointer to output data file */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                 main
Input Parameters: int argc - Number of command line arguments.
                          char **argv - pointer to command line argument string (the datafile string).
Output Parameters:        none.
Side-Effects:             Keeps track of system state and depending on current state, calls the appropriate
                 "simulate_xxxx"  functions for different l1[k] and l2[k] values, records blocking
          probability plus confidence interval statistics in output data file (pointed to by
          **argv). Also records in data file the proportion of half rate calls repacked.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void main(int argc, char **argv) {

 unsigned long NumEvents, i;       /* Number of events that have occurred so far; i is a multi-purpose counter*/
                                  /* index                                                          */
 int      seed, run;         /* Arbitrary seed for external RND NUM generator. The run var controls */
                                  /* the no. of times the simulation is performed for Conf. Int. purposes        */

 /* check command line arguments */
          if (argc != 2) {
            printf("Usage: %s data_file\n", argv[0]);
            exit(1);
          }

  /* check for output data file */
          if ((fp = fopen(argv[1], "w+t")) == NULL) {
            fprintf(stderr, "%s: Can't open %s\n", argv[0], argv[1]);
            exit(1);
          }

 printf("\n\n How many slots available for user traffic ??  ");
 scanf("%d", &Slots);

 printf("\n\n How many events to simulate ??  ");
 scanf("%d", &NumEvents);

/* printf("\n\n Seed? --> ");
 scanf("%d",&seed);           */                                  seed = 101;
 Myrand.seed(seed);

 for(run=0; run < 10; run++)

   for(prop=1, k=0; k <= 8; k++, prop++) {

    HalvesAttempted = FullsAttempted = HalvesLost = FullsLost        /* customary initilization */
    = Halves = Fulls = HalvesRepacked = 0;

    for(x=0; x < Slots; x++) {
```

```
      State[x] = 0;
      AvailFulls[x] = 0;
      AvailHalves[x] = 0; AvailHalves[(x + Slots)] = 0;
      Eligible[x] = 0;
    }

    printf("l1 = %lf ; l2 = %lf\n\n", l1[k], l2[k]);
    /* Arr. Rates must be the ones entered specifically at beginning */

    for(i=0; i<NumEvents; i++) {
      if (i%1000000 == 0) printf(".\n");
      if ((Halves==0) && (Fulls==0))              /* This code calls up the appropriate event    */
       simulate_empty();                          /* dependent on the given state of the system  */
      else if ((Halves>0) && (Fulls==0))
       simulate_halves();
      else if ((Halves==0) && (Fulls>0))
       simulate_fulls();
      else if ((Halves>0) && (Fulls>0))
       simulate_all();
      else {
       printf(" This shouldn't happen !!\n");
       exit(1);
      }
    }
   for(i=0; i<Slots; i++)
    printf("%d  ", State[i]);
   printf("\n");

   HalfBlocked[run][k] = ((double)HalvesLost / (double)HalvesAttempted);
   printf("\tHBlock = %lf",HalfBlocked[run][k]);
   SumHB[k] += HalfBlocked[run][k];
   SumHB2[k] += pow(HalfBlocked[run][k],2);

   FullBlocked[run][k] = ((double)FullsLost / (double)FullsAttempted);
   printf("\tFBlock = %lf",FullBlocked[run][k]);
   SumFB[k] += FullBlocked[run][k];
   SumFB2[k] += pow(FullBlocked[run][k],2);

   HRep[run][k] = ((double)HalvesRepacked / (double)(HalvesAttempted - HalvesLost));
   printf("\tHRep = %lf\n",HRep[run][k]);
   SumHRep[k] += HRep[run][k];
   SumHRep2[k] += pow(HRep[run][k],2);

   }                    /* End Of the Outer RUN,K LOOP STRUCTURE! */

   fprintf(fp, "REPACKING WITH RANDOM RESERVATION.  Events = %d.    Seed = %d.\n\n", NumEvents,
seed);
   fprintf(fp, "L_tot  ; \%Fulls ; HBlock ; CI_diameter ; FBlock ; CI_diameter ; \%Repacked ; CI_diameter\n\n");


/* Calculate Mean and 95% Confidence Intervals */

  for (k=0,prop=1; k<=8; k++,prop++) {
   MeanHB[k] = SumHB[k] / 10.0;
   S2HB[k] = (10.0/9.0)*((SumHB2[k]/10.0)-pow(MeanHB[k],2));
   Diam_HB[k] = 2 * ( 2.262*sqrt(S2HB[k])/sqrt(10) );
```

```
  MeanFB[k] = SumFB[k] / 10.0;
  S2FB[k] = (10.0/9.0)*((SumFB2[k]/10.0)-pow(MeanFB[k],2));
  Diam_FB[k] = 2 * ( 2.262*sqrt(S2FB[k])/sqrt(10) );

  MeanHRep[k] = SumHRep[k] / 10.0;
  VarHRep[k] = (10.0/9.0)*((SumHRep2[k]/10.0)-pow(MeanHRep[k],2));
  Diam_HRep[k] = 2 * ( 2.262*sqrt(VarHRep[k])/sqrt(10) );

  fprintf(fp, "%6.5lf;  %6.5lf;  %6.5lf;\t %6.5lf;\t %6.5lf;\t %6.5lf;\t %6.5lf;\t %6.5lf\n", (l1[k]+l2[k]), (0.1*prop),
MeanHB[k], Diam_HB[k], MeanFB[k], Diam_FB[k], MeanHRep[k], Diam_HRep[k]);
 }

}


/* Functions */
/* TYPE OF ALLOCATION : CONDITIONAL REPACKING WITH RAND. RES.                    */
/*================================================================================*/
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               HalfRateArrival
Input Parameters: none.                 Output Parameters:      none.
Side-Effects:           State[i], HalvesAttempted, Halves, and HalvesLost are modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void HalfRateArrival(void) {

   int i;                                   /* counter and index variables */
   double RandNum, P1 = 0.6;                /* P1 = Prob. Of Accepting Half, when No. = 2*slots - 2 */

   HalvesAttempted++;                       /* Register the arrival/attempt */

   if ((2*Fulls+Halves < 2*Slots - 2)||(2*Fulls+Halves == 2*Slots - 1)) {
     for (i=0; i<Slots; i++) {              /* First, work out which slots are free */
       if (State[i] < 2) {
         Halves++;
         State[i] += 1;
         break;
       }
     }
   }
   else if ( 2*Fulls+Halves == 2*Slots - 2 ) {    /* Half gets in with prob. P1 = 0.6 */
     RandNum = Myrand.uniform(0,1.0);
     if (RandNum <= 0.6) {
       for (i=0;i<Slots; i++) {             /* First, work out which slots are free */
         if (State[i] < 2) {
           Halves++;
           State[i] += 1;
           break;
         }
       }
     }
     else
      HalvesLost++;
   }
   else if (2*Fulls+Halves == 2*Slots)
    HalvesLost++;                           /* There wasn't a single half-gap anywhere. Loss occurs.   */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Function: ***HalfRateDeparture***
Input Parameters: none. Output Parameters: none.
Side-Effects: State[i], and Halves are modified as required.
~~~~~~*/

```
void HalfRateDeparture(void) {

   int i,j,RandNum,Selected;                    /* counter and index variables */

   for(i=0,j=0; i<Slots; i++) {                 /* First, work out which slots are eligible for HR depart.. */

      if (State[i] == 2) {
        AvailHalves[j] = 2*i;
        j++;
        AvailHalves[j] = 2*i + 1;
        j++;
     }
     else if (State[i] == 1) {
        AvailHalves[j] = 2*i;
        j++;
     }
   }
   RandNum = (int)Myrand.uniformInt(0,(j-1));  /* Second, have a "lottery" from which one of the available */
   Selected = AvailHalves[RandNum];       /* half rate channels departs                          */
   State[(Selected/2)] -= 1;
   Halves--;
}
```

/*~~~~~~
Function: ***FullRateArrival***
Input Parameters: none. Output Parameters: none.
Side-Effects: State[i], FullsAttempted, Fulls, and FullsLost are modified as required.
~~~~~~*/

```
void FullRateArrival(void) {

/* This is the procedure which had to be substantially modified from the FirstFit original one */

   int i,j, NoGap = 0;                          /* counter and index variables */

   FullsAttempted++;                             /* Register the arrival/attempt */

 /*   printf("\nFull Rate Arrival\n"); */
   for(i=0,j=0; i<Slots; i++) {                 /* Place in leftmost free FULL slot */

      if (State[i] == 0) {
           /*  printf("\nFull -> Totally Empty Slot, %d\n", i);*/
        Fulls++;
        State[i] = 3;
        break;
      }
      else if (State[i] == 1) {
       /*  printf("Noted -> Candidate for Rep, %d\n",i);*/
        Eligible[j] = i;
        j++;
        NoGap++;
      }
      else {
        NoGap++; /*printf("No Cigar!, State[%d] = %d\n", i, State[i]); */}
   }
```

```
    if ( (NoGap == Slots)&&(j > 1) ) {          /* There wasn't a single half-gap anywhere. Must Repack the  */

      Fulls++;                                   /* first two isolated halves (into the leftmost T/S)         */
      State[Eligible[0]] = 2;
      State[Eligible[1]] = 3;
      HalvesRepacked++;                          /* Perform the repack, increment global counter. */
      printf("Repacked: Source = %d, Dest = %d\n",Eligible[1], Eligible[0]);
    }
    else if ( (NoGap == Slots)&&(j <= 1) )
    printf("NoGap = %d, Full Rate Call BLOCKED!\n", NoGap);
    FullsLost++;                                 /* Could not get in. Bzzzt! */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:              FullRateDeparture
Input Parameters: none.                    Output Parameters:       none.
Side-Effects:          State[i] and Fulls are modified as required.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void FullRateDeparture(void) {

    int i,j,RandNum,Selected;                    /* counter and index variables */

    for(i=0,j=0; i<Slots; i++)                   /* First, work out which slots are eligible for FR departure ... */

       if (State[i] == 3) {
         AvailFulls[j] = i;
         j++;
       }

      RandNum = Myrand.uniformInt(0,(j-1)); /* Second, have a "lottery" from which one of the available       */
      Selected = AvailFulls[RandNum];       /* full rate channels departs.                          */
      State[Selected] = 0;
      Fulls--;
}

/* One of two, three or four events may occurr depending on the state the system is in */
/* An event can either be the arrival of a Full OR Half rate call, or the departure */
/* of a Full OR Half rate call. For example, state dependence is illustrated by the fact that */
/* when there are zero calls in the system, no departures can occurr, so we can only invoke */
/* the "simulate_empty()" procedure. Each invocation of it calls the RN generator with Real Number */
/* limits [0,(l1[k]+l2[k])], and determine what has happened. */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:              simulate_empty
Input Parameters: none.                    Output Parameters:       none.
Side-Effects:     Depending on randomly obtained value of X, FullRateArrival or HalfRateArrival  is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_empty(void) {

    double   X;
    X = Myrand.uniform(0,(l1[k]+l2[k]));
    if (X<l1[k])
       FullRateArrival();
    else
       HalfRateArrival();
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:              simulate_halves
```

```
Input Parameters: none.                    Output Parameters:       none.
Side-Effects:     Depending on randomly obtained value of X, FullRateArrival, HalfRateArrival  or
                  HalfRateDeparture is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_halves(void) {

    double   X;
    X = Myrand.uniform(0,(l1[k]+l2[k]+Halves*m2));
    if (X<l1[k])
      FullRateArrival();
    else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
           HalfRateArrival();
    else
      HalfRateDeparture();
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                simulate_fulls
Input Parameters: none.                    Output Parameters:       none.
Side-Effects:     Depending on randomly obtained value of X, FullRateArrival, FullRateDeparture,
                  or HalfRateArrival is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_fulls(void) {

    double   X;
    X = Myrand.uniform(0,(l1[k]+l2[k]+Fulls*m1));
    if (X<l1[k])
       FullRateArrival();
    else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
           HalfRateArrival();
    else
      FullRateDeparture();
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                simulate_all
Input Parameters: none.                    Output Parameters:       none.
Side-Effects:     Depending on randomly obtained value of X, FullRateArrival, FullRateDeparture,
                  HalfRateArrival  or  HalfRateDeparture is called.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void simulate_all(void) {

    double   X;
    X = Myrand.uniform(0,(l1[k]+l2[k]+Fulls*m1+Halves*m2));
    if (X<l1[k])
       FullRateArrival();
    else if ((X>=l1[k])&&(X<l1[k]+l2[k]))
           HalfRateArrival();
    else if ((X>=l1[k]+l2[k])&&(X<l1[k]+l2[k]+Fulls*m1))
      FullRateDeparture();
    else
      HalfRateDeparture();                  }
```

9.2 Analytic Programs

9.2.1 Fixed Boundary Scheme

```
/* Multirate Channel Allocation: Fixed Boundary Scheme - ERLANG FORMULA Method 11/9/95          */

/* Note: This is the program., dealing with 8 Full Rate time slots                              */
/*=============================================================================*/

/* Milosh V. Ivanovich , / September 1995                                                      */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define m1 0.333                    /* Avg. 3 min. holding time for both types of calls, as usual in telephony */
#define m2 0.333

/* Prototypes */

double E (double Ao, int Circuits);  /* Prototype for recursive Erlang formula function */

/* Global Variables */

double l1,l2, AFull, AHalf;                         /* The arrival rates, which will be varied          */
                                                    /* Offered half and full rate traffic, in Erlangs, obtained */
                                                    /* by multiplying arrival rate by holding time      */

double HalfBlocked[11], FullBlocked[11];            /* Arrays for full and half rate call blocking probability */

FILE    *fp;                                        /* File pointer to output data file */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                   main
Input Parameters: int argc - Number of command line arguments.
                            char **argv - pointer to command line argument string (the datafile string).
Output Parameters:          none.
Side-Effects:               By calling the E function for different l1[k] and l2[k] values, calculates and records
                            blocking probability in output data file (pointed to by **argv).
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void main(int argc, char **argv) {

  int      k;                 /* k varies the proportion of full rate calls in the traffic mix.          */
  double   p1, O;             /* Proportion of full rate calls, p1; O is the Offered Traffic per Channel */

  /* check command line arguments */
          if (argc != 2) {
            printf("Usage: %s data_file\n", argv[0]);
            exit(1);
          }

   /* check for output data file */
          if ((fp = fopen(argv[1], "w+t")) == NULL) {
            fprintf(stderr, "%s: Can't open %s\n", argv[0], argv[1]);
            exit(1);
          }
```

```
    printf("Offered Traffic per Channel =   ?");
    scanf("%lf",&O);

    for(k=0; k <= 0; k++) {

      p1 = k * 0.1;                                /* proportion of FULL RATE traffic              */
      l1 = O*8*m1 * (2*p1/(p1 + 1));               /*  given the offered traffic per channel, O,         */
      l2 = 2*(O*8*m1 - l1);                        /* hence O*8*mu is the average weighted arr. rate */

     printf("\nlambda1 = %g  ;  lambda2 = %g  ;  overall A.R. = l1+0.5*l2 = %g\n\n", l1,l2,(l1+0.5*l2));

      AFull = l1 / (m1);                           /* Calculate the Offered Traffic in Erlangs */
      AHalf = l2 / (m1);                           /* NOTE: Arbitrary 50% / 50% FIXED BOUNDARY, so
                                                   /* that slots are split evenly between half/full rate */
                                                   /* channels                                    */

      /* Blocking Probability Calculation */

      HalfBlocked[k] = E(AHalf, 8);          /* 8 H/R Circuits are available */
      FullBlocked[k] = E(AFull, 4);          /* 4 F/R Circuits are available */

      printf(" The Probability of Blocking is:  Full Rate Calls       Half Rate Calls ");
      printf("\n -------------------------------------------------------------------");
      printf("\n\t\t\t\t     %lf            %lf           \n\n", FullBlocked[k], HalfBlocked[k]);

      fprintf(fp, "%g\t\t\t %g\n", FullBlocked[k], HalfBlocked[k]);
    }
  }

/* Functions */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                 E
Input Parameters: double Ao - Offered Traffic in Erlangs.
                          int Circuits - pointer to command line argument string (the datafile string).
Output Parameters:        double E - probability of congestion.
Side-Effects:             Recursively works out Erlang loss formula and returns value to main.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
double E (double Ao, int Circuits) {

   double Etemp = 1.0;
   int    i;
   for (i=1; i<=Circuits; i++)
    Etemp = (Ao*Etemp)/(i + Ao*Etemp);
   return Etemp;
}
```

9.2.2 Sliding Boundary Scheme

```
/* Multirate Channel Allocation: Sliding Boundary Scheme- ERLANG FORMULA Method _____ 11/9/95 */

/* Note: This is the prog. dealing with 8 Full Rate timeslots                                */
/*========================================================================= */

/* Milosh V. Ivanovich , September 1995                                                      */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define m1 0.333          /* Avg. 3 min. holding time for both types of calls, as usual in telephony */
#define m2 0.333

/* Prototypes */

double E (double Ao, int Circuits);          /* Prototype for recursive Erlang formula function */

/* Global Variables */

double l1,l2, AFull, AHalf;                  /* The arrival rates, which will be varied ...        */
                                             /* Offered half and full rate traffic, in Erlangs, obtained */
                                             /* by multiplying arrival rate by holding time        */

double HalfBlocked[11], FullBlocked[11];     /* Arrays for full and half rate call blocking probability */
FILE    *fp;                                 /* File pointer to output data file */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               main
Input Parameters: int argc - Number of command line arguments.
                        char **argv - pointer to command line argument string (the datafile string).
Output Parameters:      none.
Side-Effects:           By calling the E function for different l1[k] and l2[k] values, calculates and records
                        blocking probability in output data file (pointed to by **argv).
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void main(int argc, char **argv) {

  int      k, FullCircuits;     /* k varies the proportion of fulls in traffic mix. FullCircuits gives the size of */
                                /* reserved space for full rate circuits                         */
  double   p1, O;               /* Proportion of full rate calls, p1; O is the Offered Traffic per Channel      */

  /* check command line arguments */
        if (argc != 2) {
          printf("Usage: %s data_file\n", argv[0]);
          exit(1);
        }

  /* check for output data file */
        if ((fp = fopen(argv[1], "w+t")) == NULL) {
          fprintf(stderr, "%s: Can't open %s\n", argv[0], argv[1]);
          exit(1);
        }

   /*  printf("Offered Traffic per Channel = ");
```

```
   scanf("%lf",&O);   */

   for(k=0; k <= 0; k++) {

     p1 = k * 0.1;                                    /* proportion of FULL RATE traffic            */
     l1 = O*8*m1 * (2*p1/(p1 + 1));                   /*  given the offered traffic per channel, O,   */
     l2 = 2*(O*8*m1 - l1);                            /* hence O*8*mu is the average weighted arr. rate */

    printf("\nlambda1 = %g  ;  lambda2 = %g  ;  overall A.R. = l1+0.5*l2 = %g\n\n", l1,l2,(l1+0.5*l2));

     AFull = l1 / (m1);                       /* Calculate the Offered Traffic in Erlangs */
     AHalf = l2 / (m1);                       /* NOTE: SLIDING BOUNDARY employed now */
                                              /* slots are reserved unevenly for half and full rate channels */

     /* Blocking Probability Calculation */

     printf("Full Circuits Allocated Dynamically = %d\n", FullCircuits = int(8*p1));

     HalfBlocked[k] = E(AHalf, 2*(8-FullCircuits));    /* (8-FullCircuits) are available for H/R only */
     FullBlocked[k] = E(AFull, FullCircuits); /* FullCircuits are available for F/R only       */

     printf(" The Probability of Blocking is:  Full Rate Calls       Half Rate Calls ");
     printf("\n -------------------------------------------------------------------");
     printf("\n\t\t\t\t     %lf            %lf           \n\n", FullBlocked[k], HalfBlocked[k]);

     fprintf(fp, "%g\t\t\t %g\n", FullBlocked[k], HalfBlocked[k]);
   }
 }

/* Functions */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                 E
Input Parameters: double Ao - Offered Traffic in Erlangs.
                          int Circuits - pointer to command line argument string (the datafile string).
Output Parameters:        double E - probability of congestion.
Side-Effects:             Recursively works out Erlang loss formula and returns value to main.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
double E (double Ao, int Circuits) {

  double Etemp = 1.0;
  int    i;
  for (i=1; i<=Circuits; i++)
   Etemp = (Ao*Etemp)/(i + Ao*Etemp);
  return Etemp;
}
```

9.2.3 Analytic Optimisation Problem Solver Program

```
/*                          The "SOLV_##.C" Series:                          */
/*                  An Automated State-Space Formulation and Solution Program  */
/*                  Version 2: Optimisation Problem - Solution Framework       */

/*          Milosh V. Ivanovich, October 1995.                               */

/* NOTES:                                                                    */
/* ----------                                                                */
/* This program, in its original version obtained three inputs from the user, namely,  */
/* (a) Frame Size in Number of Slots; (b) Packing Scheme; and (c) Arrival and          */
/* Departure Rates.                                                          */

/* In this version, the program has been augmented in order to facilitate the solving  */
/* of an optimisation problem.  User now only specifies the (a) Packing Scheme, and the */
/* (b) Frame Size in Number of Slots. The program then obtains Maximum Permitted       */
/* Total Customer Arrival rate, constrained by a 2% blocking probability for both types */
/* of user.                                                                  */

/* It automatically sets up the Q-Matrix in ordinary form, then manipula-             */
/* tes it into Q', where it is in the form p = p . Q' and finally solves              */
/* it using the method of Successive Relaxation.                             */

/* Normalises the p[n]s ONLY after all iterations (preferable)                */
/* Assumption: Rigid walls between Full Rate channels                        */

/* p[n]     ==> The probability of being in state n                          */
/*                  Ultimately we are after p[n] for all possible values of n.  */

/* Q[n1][n2]  ==> The (n1, n2)-th element of the State Coefficient Matrix     */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* Prototypes */

void   body(void);
void   q_rnd(void);
void   q_bestfit(void);
void   q_repacking(void);
void   q_rpck_perp_reservation(void);
void   q_rpck_rnd_reservation(void);
void   q_rpck_perp_hs_reservation(void);
void   manipulate(void);
void   record_q(void);
void   record_results(void);

/* Global defines */

#define m1 0.333                /* Avg. 3 min. holding time for both types of calls, as usual in telephony */
#define m2 0.333
#define TRUE 1
#define FALSE 0
#define E 1E-5
```

```
#define K 3000              /* IMPORTANT: PC Version cannot support more than a 100 x 100      */
                            /* P matrix of "floats" due to memory constraints                   */
                            /* Hence, program relies on Unix SunSparc platform testbed.         */

#define p1 0.6 /* Only applicable for RRR Algorithm.                                            */
#define p2 1.0 /* p1, p2: The probabilities that a half rate call is accepted                  */
               /* when we have [2*Slots-2] or [2*Slots-1] occupied places, respectively.       */

/* Global data structures */

float *p                    ;  /* P(state occupancy) array            */
float *pold                 ;  /* Previous iteration of p[n]          */
float Q[K][K]               ;  /* The Generated Q Matrix              */

int HalfBlocked[K]                  ;  /* Arrays keeping track of which states     */
int HalfBlk_p1[K], HalfBlk_p2[K]    ;  /* Semi-Blocked State Trackers                   */
int FullBlocked[K]          ;  /* are blocking for the two types of calls       */
int halfcount=0, fullcount=0        ;  /* Global counters for this purpose              */
int Halves, Fulls, IsolHalves       ;         /* System counter for given types of packing   */
                                              /* - effectively defines state                  */

int     *ChkArr             ;       /* Array of True/False values, to              */
                                    /* signify whether the difference              */
                                    /* between the old and current                 */
                                    /* iterations is less than E                   */
int     Stop = 0            ;       /* Stop is only set to 1,                      */
                                    /* when all ChkArr booleans are                */
                                    /* set to 1.                                   */
int     Xmax, Ymax          ;       /* Q Matrix Size Markers                       */
int     x = -1, y = -1      ;       /* Q Matrix Global indices                     */
int     Slots               ;       /* User Sets this. State Limited on            */
                                    /* most computers. Must not exceed 8           */
float l1[11],l2[11], RowSum ;       /* The arrival rates, which will be            */
                                    /* given by the user ...                       */
float HRBlocking, FRBlocking ;      /* Global variables for blocking probabilities */
FILE   *fp                  ;       /* Global File Pointer                         */

int    method, I=0, z=0     ;            /* Which packing method, user response. */
                                         /* I is a counter var. z varies the prop. of full rate calls */
float  R,rho, prop1, TempSum, BigSum=0  ;  /* Temporary Sum variables TempSum and BigSum */
                                         /* prop1 is the proportion  of full rate callers          */
                                         /* R and rho are the starting point and increment in the */
                                         /* optimisation algortihm                                */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:              main
Input Parameters: int argc - Number of command line arguments.
                       char **argv - pointer to command line argument string (the datafile string).
Output Parameters:     none.
Side-Effects:          User-specified function from the suite of "qgen_xxxx" functions is called to generate
                       the relevant Q matrix for desired scheme. Optimisation is then carried by means of
                       three separate sets of double loops within main() itself out to find the maximum
                 allowable number of user arrivals, under the Grade of Service constraint. The
                 equations resulting from the Q matrix are solved by Gauss-Seidel iteration, when the
                 body() function is called within these double-loops.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void main(int argc, char **argv) {
```

```
/* check command line arguments */
    if (argc != 2) {
     printf("Usage: %s data_file\n", argv[0]);
     exit(1);
    }

/* check for output data file */
    if ((fp = fopen(argv[1], "w+t")) == NULL) {
     fprintf(stderr, "%s: Can't open %s\n", argv[0], argv[1]);
     exit(1);
    }

printf(" WELCOME TO THE AUTOMATED STATE-SPACE FORMULATION/SOLUTION
PROGRAM.\n");
printf("=========================================================\n\n");
printf("(A) Number Of Slots Per Frame? > ");
scanf("%d", &Slots);
printf("\n");

printf("(B) Choose Packing Discipline:\n");
printf("------------------------------\n\n");
printf("1. Random                    2. Best Fit\n\n");
printf("3. Repacking                 4. Repacking with Perpetual Reservation (RPR)\n\n");
printf("5. Repacking with Random Reservation (RRR)\n\n ");
printf("6. Repacking with Forced Blocking Equalisation (RPHSR)\n\n> ");
scanf("%d",&method);

fprintf(fp,"Number of Slots in Frame = %d\n", Slots);

 if (method==1) fprintf(fp, "Packing: Random\n");
 else if (method==2) fprintf(fp, "Packing: Best Fit\n");
 else if (method==3) fprintf(fp, "Packing: Repacking\n");
 else if (method==4) fprintf(fp, "Packing: Repacking with Perpetual Reservation\n");
 else if (method==5) fprintf(fp, "Packing: Repacking with Random Reservation\n");
 else if (method==6) fprintf(fp, "Packing: Repacking with Perpetual Half Slot Reservation\n");


 fprintf(fp, "\nWhat follows is a list of Maximized Lambda_Total values (i.e. Max. Capacity)\n");
 fprintf(fp, "at varying traffic mixes, subject to the constraint that neither P(Block.) exceeds 2%\n");
 fprintf(fp, "\nFull Rate Prop. ; Lambda Tot ; H.R. Blocking Prob. ; F.R. Blocking Prob. ; Rho\n");

 z = prop1 = 0;
 for(rho = 0.74; (HRBlocking <0.0180); rho+=0.01)          /* OPTIMIZE loop, conditioned by prob. */
  body();

 printf("Domah!\n");
 fprintf(fp, "%6.6f\t\t %6.6f\t\t", prop1, (l1[z]+l2[z]));
 fprintf(fp, "  %6.6f\t    %6.6f\t   ; at rho = %4.3f\n", HRBlocking, FRBlocking,  ((l1[z]+0.5*l2[z])/(Slots*m1)));

for(z = 1, R = 0.65; z<=9; z++) {                          /* Varying the traffic mix. */

 prop1 = z * 0.1;                                          /* proportion of FULL RATE traffic */
 if (z%2 == 0) R-=0.02;                                    /* incremental factor */
 HRBlocking = FRBlocking = 0;
 for(rho = R; (HRBlocking <0.0195)&&(FRBlocking < 0.0195); rho+=0.002) { /* OPTIMIZE loop,
                                                                          conditioned by prob. */
  body();
```

```
  printf("%6.6f\t\t %6.6f\t\t", prop1, (l1[z]+l2[z]));
  printf("  %6.6f\t    %6.6f\t   ; at rho = %4.3f\n", HRBlocking, FRBlocking, ((l1[z]+0.5*l2[z])/(Slots*m1)));
 }
 printf("Domah!\n");
 fprintf(fp, "%6.6f\t\t %6.6f\t\t", prop1, (l1[z]+l2[z]));
 fprintf(fp, "  %6.6f\t    %6.6f\t   ; at rho = %4.3f\n", HRBlocking, FRBlocking, ((l1[z]+0.5*l2[z])/(Slots*m1)));
}

 z = 10;
 prop1 = 1;
 HRBlocking = FRBlocking = 0;

 for(rho = 0.58; (FRBlocking <0.0195); rho+=0.002)   /* OPTIMIZE loop, conditioned by prob. */
  body();
 printf("Domah!\n");
 fprintf(fp, "%6.6f\t\t %6.6f\t\t", prop1, (l1[z]+l2[z]));
 fprintf(fp, "  %6.6f\t    %6.6f\t   ; at rho = %4.3f\n", HRBlocking, FRBlocking, ((l1[z]+0.5*l2[z])/(Slots*m1)));

 fclose(fp);
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/* Functions */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:               q_rnd
Input Parameters:       none.
Output Parameters:      none.
Side-Effects:           Generates the Q matrix, storing elements into the global Q[x][y] variable, using the
                        Random channel allocation scheme. Prompts user to record matrix elements by
                        calling record_q().
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void q_rnd(void) {

 long int     i,j,k;                         /* Various Indices  */
 int          InitOuter, InitInner;          /* Odd/Even Flags   */

     for(Fulls = 0; Fulls<=Slots; Fulls++)                          /* Going Down the Rows */
      for(Halves = 0; Halves <= 2*(Slots - Fulls); Halves++) {
          if ( Halves%2==0 ) InitOuter = 0; else InitOuter = 1;
          for(IsolHalves = InitOuter; (IsolHalves <= Halves) &&
             (IsolHalves + Halves + 2*Fulls <= 2*Slots); IsolHalves+=2) {

          /* Notes: (i) Halves counter cannot exceed the limit imposed by the number */
          /*            of available minus number of occupied "half slots"           */
          /*        (ii) InitOuter starts at zero when we have an EVEN no. of Halves */
          /*        (iii) IsolHalves, intuitively, must be constrained by three things */
          /*         the number of halves, whether there ARE halves at all, and */
          /*              the TOTAL FREE half slots in the frame                     */

          x++;
          RowSum = 0;    /* "Zero" both the Col. counter AND Row-Sum */
          y = -1;

     /* Now, determine the BLOCKING STATES !                   */

          if ((Slots - Fulls - IsolHalves - 0.5*(Halves-IsolHalves)) < 1) {
           FullBlocked[fullcount] = x;
           fullcount++;
```

```
        }

        if ((2*Fulls + Halves) == (2*Slots)) {
         HalfBlocked[halfcount] = x;
         halfcount++;
        }

         for(i = 0; i<=Slots; i++)                                    /* Going across the Columns */
          for(j = 0; j <= 2*(Slots - i); j++) {
           if ( j%2==0 ) InitInner = 0; else InitInner = 1;
           for(k = InitInner; (k <= j)&&(k <= 2*(Slots-i) - j);k+=2) {
            y++;
            if (x!=y) {                                                  /* Diagonal must be done last */

             if ((i-Fulls==1)&&(j==Halves)&&(k==IsolHalves)) {                  /* Full ARRIVAL */
              RowSum += Q[x][y] = l1[z];
             }
             else if ((i-Fulls==-1)&&(j==Halves)&&(k==IsolHalves)) {            /* Full DEPARTURE */
              RowSum += Q[x][y] = Fulls*m1;
             }
             else if ((i==Fulls)&&(j-Halves==1)&&(k-IsolHalves==1)) {   /* Half Rate ARRIVAL Type 1a/b */
                    if (IsolHalves == 0)                                 /* These arrivals CREATE gaps! */
                     RowSum += Q[x][y] = l2[z];
                    else
                     RowSum += Q[x][y] =
                      l2[z]*(float)(2*Slots-2*Fulls-Halves-IsolHalves)/(float)(2*Slots-2*Fulls-Halves);
             }
             else if ((i==Fulls)&&(j-Halves==1)&&(k-IsolHalves==-1)) { /* Half Rate ARRIVAL Type 2a/2b/*
                    if (2*Slots-2*Fulls-Halves == IsolHalves)            /* These arrivals FILL IN gaps! */
                     RowSum += Q[x][y] = l2[z];
                    else
                     RowSum += Q[x][y] =
                      l2[z]*(float)(IsolHalves)/(float)(2*Slots-2*Fulls-Halves);
             }
             else if ((i==Fulls)&&(j-Halves==-1)&&(k-IsolHalves==1)) {  /* Half Rate DEPARTURE Type
                                                                          1a/b*/
                    if (IsolHalves == 0)                          /* These departures CREATE gaps!  */
                     RowSum += Q[x][y] = Halves*m2;
                    else
                     RowSum += Q[x][y] =
                      Halves*m2*(float)(Halves-IsolHalves)/(float)(Halves);
             }
             else if ((i==Fulls)&&(j-Halves==-1)&&(k-IsolHalves==-1)) {    /* Half Rate DEPARTURE Type
                                                                            2a/2b */
                    if (IsolHalves == Halves)                     /* These departures FILL IN gaps! */
                     RowSum += Q[x][y] = Halves*m2;
                    else
                     RowSum += Q[x][y] =
                      Halves*m2*(float)(IsolHalves)/(float)(Halves);
             }
             else Q[x][y] = 0;
            }
           }
          }
          Q[x][x] = - RowSum;
        }
}
```

```
     printf("\nMatrix Size Check (Dimensions x by y): x =");
     printf(" %d, y = %d \n\n", Xmax = x, Ymax = y);

    record_q();   /* If user wishes, record Q-Matrix to file. */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                  q_bestfit
Input Parameters:  none.
Output Parameters:         none.
Side-Effects:              Generates the Q matrix, storing elements into the global Q[x][y] variable, using the
                           Best Fit channel allocation scheme. Prompts user to record matrix elements by
                           calling  record_q().
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void q_bestfit(void) {

 long int     i,j,k;                              /* Various Indices  */
 int            InitOuter, InitInner;             /* Odd/Even Flags   */

     for(Fulls = 0; Fulls<=Slots; Fulls++)                      /* Going Down the Rows */
      for(Halves = 0; Halves <= 2*(Slots - Fulls); Halves++) {
            if ( Halves%2==0 ) InitOuter = 0; else InitOuter = 1;
            for(IsolHalves = InitOuter; (IsolHalves <= Halves) &&
                (IsolHalves + Halves + 2*Fulls <= 2*Slots); IsolHalves+=2) {

            x++;
            RowSum = 0;    /* "Zero" both the Col. counter AND Row-Sum */
            y = -1;

       /* Now, determine the BLOCKING STATES !                  */

            if ((Slots - Fulls - IsolHalves - 0.5*(Halves-IsolHalves)) < 1) {
             FullBlocked[fullcount] = x;
             fullcount++;
            }

            if ((2*Fulls + Halves) == (2*Slots)) {
             HalfBlocked[halfcount] = x;
             halfcount++;
            }

             for(i = 0; i<=Slots; i++)                                /* Going across the Columns */
              for(j = 0; j <= 2*(Slots - i); j++) {
               if ( j%2==0 ) InitInner = 0; else InitInner = 1;
               for(k = InitInner; (k <= j)&&(k <= 2*(Slots-i) - j);k+=2) {
                y++;
                if (x!=y) {                                               /* Diagonal must be done last */

                 if ((i-Fulls==1)&&(j==Halves)&&(k==IsolHalves)) {                /* Full ARRIVAL */
                   RowSum += Q[x][y] = l1[z];
                 }
                 else if ((i-Fulls==-1)&&(j==Halves)&&(k==IsolHalves)) {     /* Full DEPARTURE */
                   RowSum += Q[x][y] = Fulls*m1;
                 }
                 else if ((i==Fulls)&&(j-Halves==1)&&(k-IsolHalves==1)&&
                                    (IsolHalves==0)) {                    /* Half Rate ARRIVAL Type 1a/1b */
                   RowSum += Q[x][y] = l2[z];                             /* These arrivals CREATE gaps! */
                 }
```

```
            else if ( (i==Fulls)&&(j-Halves==1)&&(k-IsolHalves==-1)&&
                        (IsolHalves>0) ) {                  /* Half Rate ARRIVAL Type 2a/2b */
                 RowSum += Q[x][y] = l2[z];                 /* These arrivals FILL IN gaps! */
            }
            else if ((i==Fulls)&&(j-Halves==-1)&&(k-IsolHalves==1)) {    /* Half Rate DEPARTURE Type
                                                                          1a/b */
                 if (IsolHalves == 0)                       /* These departures CREATE gaps!  */
                  RowSum += Q[x][y] = Halves*m2;
                 else
                  RowSum += Q[x][y] =
                   Halves*m2*(float)(Halves-IsolHalves)/(float)(Halves);
            }
            else if ((i==Fulls)&&(j-Halves==-1)&&(k-IsolHalves==-1)) {    /* Half Rate DEPARTURE Type
                                                                          2a/2b */
                 if (IsolHalves == Halves)                  /* These departures FILL IN gaps! */
                  RowSum += Q[x][y] = Halves*m2;
                 else
                  RowSum += Q[x][y] =
                   Halves*m2*(float)(IsolHalves)/(float)(Halves);
            }
            else Q[x][y] = 0;
           }
          }
         }
         Q[x][x] = - RowSum;
        }
     }
    printf("\nMatrix Size Check (Dimensions x by y): x =");
    printf(" %d, y = %d \n\n", Xmax = x, Ymax = y);

    record_q();   /* If user wishes, record Q-Matrix to file. */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Function: ***q_repacking***

Input Parameters: none.

Output Parameters: none.

Side-Effects: Generates the Q matrix, storing elements into the global Q[x][y] variable, using the Repacking channel allocation scheme. Prompts user to record matrix elements by calling ***record_q()***

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void q_repacking(void) {

 long int     i,j;                    /* Various Indices  */

     for(Fulls = 0; Fulls<=Slots; Fulls++)                  /* Going Down the Rows */
      for(Halves = 0; Halves <= 2*(Slots - Fulls); Halves++) {

           x++;
           RowSum = 0;    /* "Zero" both the Col. counter AND Row-Sum */
           y = -1;

       /* Now, determine the BLOCKING STATES !                */

           if ((2*Fulls + Halves) > 2*(Slots-1)) {
            FullBlocked[fullcount] = x;
            fullcount++;
           }
```

```
            if ((2*Fulls + Halves) > (2*Slots - 1)) {
             HalfBlocked[halfcount] = x;
             halfcount++;
            }

            for(i = 0; i<=Slots; i++)                    /* Going across the Columns */
             for(j = 0; j <= 2*(Slots - i); j++) {

               y++;
               if (x!=y) {                                  /* Diagonal must be done last */

                if ((i-Fulls==1)&&(j==Halves)) {            /* Full ARRIVAL */
                  RowSum += Q[x][y] = l1[z];
                }
                else if ((i-Fulls==-1)&&(j==Halves)) {   /* Full DEPARTURE */
                  RowSum += Q[x][y] = Fulls*m1;
                }
                else if ((i==Fulls)&&(j-Halves==1)) {    /* Half Rate ARRIVAL */
                  RowSum += Q[x][y] = l2[z];
                }
                else if ((i==Fulls)&&(j-Halves==-1)) {    /* Half Rate DEPARTURE */
                 RowSum += Q[x][y] = Halves*m2;
                }
                else Q[x][y] = 0;
               }
             }
            Q[x][x] = - RowSum;
      }

     printf("\nMatrix Size Check (Dimensions x by y): x =");
     printf(" %d, y = %d \n\n", Xmax = x, Ymax = y);

    record_q();   /* If user wishes, record Q-Matrix to file. */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                  q_rpck_perp_reservation
Input Parameters: none.
Output Parameters:         none.
Side-Effects:              Generates the Q matrix, storing elements into the global Q[x][y] variable, using the
                           RPR Fit channel allocation scheme. Prompts user to record matrix elements by
                           calling  record_q()
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void q_rpck_perp_reservation(void) {

 long int     i,j;                      /* Various Indices  */

      for(Fulls = 0; Fulls<=Slots; Fulls++)                        /* Going Down the Rows */
       for(Halves = 0; Halves <= 2*(Slots - Fulls); Halves++) {

            x++;
            RowSum = 0;    /* "Zero" both the Col. counter AND Row-Sum */
            y = -1;

       /* Now, determine the BLOCKING STATES !                    */

            if ((2*Fulls + Halves) > 2*(Slots-1)) {
```

```
            FullBlocked[fullcount] = x;
            fullcount++;
           }

       if ((2*Fulls + Halves) >= 2*(Slots-1)) {
            HalfBlocked[halfcount] = x;
            halfcount++;
           }

           for(i = 0; i<=Slots; i++)                /* Going across the Columns */
            for(j = 0; j <= 2*(Slots - i); j++) {

              y++;
              if (x!=y) {                           /* Diagonal must be done last */

               if ((i-Fulls==1)&&(j==Halves)) {     /* Full ARRIVAL */
                 RowSum += Q[x][y] = l1[z];
               }
               else if ((i-Fulls==-1)&&(j==Halves)) {   /* Full DEPARTURE */
                 RowSum += Q[x][y] = Fulls*m1;
               }
               else if ( (i==Fulls)&&(j-Halves==1)&&
                        (2*Fulls+Halves < (2*Slots - 2)) ) { /* Half Rate ARRIVAL */
                 RowSum += Q[x][y] = l2[z];
               }
               else if ((i==Fulls)&&(j-Halves==-1)) {    /* Half Rate DEPARTURE */
                RowSum += Q[x][y] = Halves*m2;
               }
               else Q[x][y] = 0;
              }
            }
           Q[x][x] = - RowSum;
      }

    printf("\nMatrix Size Check (Dimensions x by y): x =");
    printf(" %d, y = %d \n\n", Xmax = x, Ymax = y);

   record_q();   /* If user wishes, record Q-Matrix to file. */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Function: ***q_rpck_rnd_reservation***
Input Parameters: none.
Output Parameters: none.
Side-Effects: Generates the Q matrix, storing elements into the <u>global Q[x][y] variable,</u> using the RRR Fit channel allocation scheme. Prompts user to record matrix elements by calling ***record_q()***

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void q_rpck_rnd_reservation(void) {

 long int     i,j;                    /* Various Indices  */

     for(Fulls = 0; Fulls<=Slots; Fulls++)                 /* Going Down the Rows */
      for(Halves = 0; Halves <= 2*(Slots - Fulls); Halves++) {

           x++;
           RowSum = 0;    /* "Zero" both the Col. counter AND Row-Sum */
           y = -1;
```

```
  /* Now, determine the BLOCKING STATES !                    */

       if ((2*Fulls + Halves) > 2*(Slots-1)) {   /* Full Rate Calls */
        FullBlocked[fullcount] = x;
        fullcount++;
       }

       if ((2*Fulls + Halves) == 2*(Slots-1)) {   /* Half Rate Calls */
        HalfBlocked[halfcount] = x;                        /* Blocked with Pr. = 1-p1 */
        HalfBlk_p1[halfcount] = x;
        halfcount++;
       }
       else if ((2*Fulls + Halves) == (2*Slots-1)) {   /* Half Rate Calls */
        HalfBlocked[halfcount] = x;                          /* Blocked with Pr. = 1-p2 */
        HalfBlk_p2[halfcount] = x;
        halfcount++;
       }
       else if ((2*Fulls + Halves) == 2*Slots) {       /* Half Rate Calls */
        HalfBlocked[halfcount] = x;                          /* With 100% prob. */
        HalfBlk_p1[halfcount] = HalfBlk_p2[halfcount] = -1;   /* Marker */
        halfcount++;
       }

       for(i = 0; i<=Slots; i++)                    /* Going across the Columns */
        for(j = 0; j <= 2*(Slots - i); j++) {

         y++;
         if (x!=y) {                                          /* Diagonal must be done last */

          if ((i-Fulls==1)&&(j==Halves)) {                 /* Full ARRIVAL */
            RowSum += Q[x][y] = l1[z];
          }
          else if ((i-Fulls==-1)&&(j==Halves)) {   /* Full DEPARTURE */
            RowSum += Q[x][y] = Fulls*m1;
          }
          else if ( (i==Fulls)&&(j-Halves==1) ) {  /* Half Rate ARRIVAL */
           if (2*Fulls+Halves < 2*(Slots - 1))     /* Not needing the reserved slot. */
            RowSum += Q[x][y] = l2[z];
           else if (2*Fulls+Halves == 2*(Slots - 1))
            RowSum += Q[x][y] = l2[z]*p1;                /* Weighted by prob. of transition. */
           else if (2*Fulls+Halves == (2*Slots - 1))
            RowSum += Q[x][y] = l2[z]*p2;            /* Weighted by prob. of transition. */
          }
          else if ((i==Fulls)&&(j-Halves==-1)) {    /* Half Rate DEPARTURE */
           RowSum += Q[x][y] = Halves*m2;
          }
          else Q[x][y] = 0;
         }
        }
       Q[x][x] = - RowSum;
  }

printf("\nMatrix Size Check (Dimensions x by y): x =");
printf(" %d, y = %d \n\n", Xmax = x, Ymax = y);

record_q();   /* If user wishes, record Q-Matrix to file. */
```

```
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:           q_rpck_perp_hs_reservation
Input Parameters:   none.
Output Parameters:  none.
Side-Effects:       Generates the Q matrix, storing elements into the global Q[x][y] variable, using the
                    RPHSR Fit channel allocation scheme. Prompts user to record matrix elements by
                    calling  record_q()
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void q_rpck_perp_hs_reservation(void) {

 long int    i,j;                /* Various Indices  */

    for(Fulls = 0; Fulls<=Slots; Fulls++)                 /* Going Down the Rows */
     for(Halves = 0; Halves <= 2*(Slots - Fulls); Halves++) {

          x++;
          RowSum = 0;   /* "Zero" both the Col. counter AND Row-Sum */
          y = -1;

     /* Now, determine the BLOCKING STATES !                  */

          if ((2*Fulls + Halves) > 2*(Slots-1)) {
           FullBlocked[fullcount] = x;
           fullcount++;
          }

     if ((2*Fulls + Halves) > 2*(Slots-1)) {
           HalfBlocked[halfcount] = x;
           halfcount++;
          }

          for(i = 0; i<=Slots; i++)              /* Going across the Columns */
           for(j = 0; j <= 2*(Slots - i); j++) {

             y++;
             if (x!=y) {                               /* Diagonal must be done last */

              if ((i-Fulls==1)&&(j==Halves)) {         /* Full ARRIVAL */
                RowSum += Q[x][y] = l1[z];
              }
              else if ((i-Fulls==-1)&&(j==Halves)) {   /* Full DEPARTURE */
                RowSum += Q[x][y] = Fulls*m1;
              }
              else if ( (i==Fulls)&&(j-Halves==1)&&
                        (2*Fulls+Halves < (2*Slots - 1)) ) { /* Half Rate ARRIVAL */
                RowSum += Q[x][y] = l2[z];
              }
              else if ((i==Fulls)&&(j-Halves==-1)) {    /* Half Rate DEPARTURE */
               RowSum += Q[x][y] = Halves*m2;
              }
              else Q[x][y] = 0;
             }
           }
          Q[x][x] = - RowSum;
    }
```

```
    printf("\nMatrix Size Check (Dimensions x by y): x =");
    printf(" %d, y = %d \n\n", Xmax = x, Ymax = y);

    record_q();   /* If user wishes, record Q-Matrix to file. */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                manipulate
Input Parameters: none.
Output Parameters:       none.
Side-Effects:            Re-arranges elements of  global Q[x][y] array (matrix) into Q' thereby obtaining the
                         form of p = p.Q' which is solveable.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void manipulate(void) {

   int i,j;
   static float temp[K][K];

   for(j=0; j<=Xmax; j++) {
      for(i=0; i<=Ymax; i++)
            temp[i][j] = Q[i][j] / (-Q[j][j]);
      temp[j][j] = 0;
   }
   for(i=0; i<=Xmax; i++)
    for(j=0; j<=Ymax; j++)
     Q[i][j] = temp[i][j];
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Function:                record_q
Input Parameters: none.
Output Parameters:       none.
Side-Effects:            If the user wishes, this procedure records elements of the global Q[x][y] array
                         (matrix) into a file, the name of which is supplied by user.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void record_q(void) {

 FILE   *fptr;                /*  File pointer */
 char   filename[20];         /* String with Filename information */
 int        done = 0;
 char   answer[5];            /* Record Q or Not? */

 while ( !done ) {

     printf("\n(D) Record the contents of the Q-Matrix? (y/n) > ");
     scanf("%s", answer);
     done = 1;
     if ((answer[0] == 'n') || (answer[0] == 'N')) {
      return;
     }
     else if ((answer[0] != 'y') && (answer[0] != 'Y')) {
      done = 0;
     }
 }

 printf("\nQ-Matrix Output Filename > ");
 scanf("%s",filename);

 fptr = fopen(filename,"w+t");
```

```
 for(x = 0; x <= Xmax; x++) {
  for(y = 0; y <= Ymax; y++)
  fprintf(fptr, "%3.4f ", Q[x][y]);
   fprintf(fptr,"\n");
  }

 fprintf(fptr,"\n\n");                      /* As a check, this prints out the valid state space */
 for(Fulls = 0; Fulls<=Slots; Fulls++)
  for(Halves = 0; Halves <= 2*(Slots - Fulls); Halves++)
   fprintf(fptr,"%d, %d\n", Fulls, Halves);
 fclose(fptr);
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Function: ***record_results***

Input Parameters: none.

Output Parameters: none.

Side-Effects: Prints to a file pointed to by the *****argv*** character string of the ***main()***, a tabulated form of results, with the Total No. of Arrivals, Proportion of Fulls, and the Half and Full rate Blocking Probabilities.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void record_results(void) {

  int i;                    /* Counter Variable */
  float TempSum = 0;        /* Temporary Sum variable */

   for(i=0; i<halfcount; i++) {
   if (method != 5) {
    TempSum += p[HalfBlocked[i]];
   }
   else if (HalfBlocked[i] == HalfBlk_p1[i]) {
    TempSum += p[HalfBlocked[i]] * (1-p1);
   }
   else if (HalfBlocked[i] == HalfBlk_p2[i]) {
    TempSum += p[HalfBlocked[i]] * (1-p2);
   }
   else {
    TempSum += p[HalfBlocked[i]];
   }
  }
  HRBlocking = TempSum;          /* Prints H.R. Blocking Prob. on the line */

  TempSum = 0;                   /* Reset, for next task. */

  for(i=0; i<fullcount; i++) {
   TempSum += p[FullBlocked[i]];
  }
  FRBlocking = TempSum;       /* Prints F.R. Blocking Prob. on same line */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Function: ***body***

Input Parameters: none.

Output Parameters: none.

Side-Effects: Depending on user input, (i.e. value of global method variable), this function calls the appropriate Q-matrix generator function (e.g. ***q_rnd()*** or ***q_rpck()***). Then calls the ***manipulate()*** function, after which Gauss-Seidel iteration is used to repeatedly modify global probability of state vars. p[x] and pold[x], until convergence is reached and all probabilities of state are solved..

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
void body(void) {

   int i,j;

   l1[z] = rho*Slots*m1 * (2*prop1/(prop1 + 1));    /* given the average total util of 0.45, hence 0.45*8*mu */
   l2[z] = 2*(rho*Slots*m1 - l1[z]);                /* is the average arr. rate, we work out l1[z]          */

   Halves = Fulls = IsolHalves = 0;           /* Zero State-Counters */
   fullcount = halfcount = 0;
   x = y = -1;

  if (method == 1) {        /* Invoke the Q-Matrix Generator Function */
   q_rnd();
  }
  else if (method == 2) {
   q_bestfit();
  }
  else if (method == 3) {
   q_repacking();
  }
  else if (method == 4) {
   q_rpck_perp_reservation();
  }
  else if (method == 5) {
   q_rpck_rnd_reservation();
  }
  else if (method==6)
   q_rpck_perp_hs_reservation();

   /*printf("Obtained Q Successfully");*/
   manipulate();           /* Arrange the equations into p . Q' = p form */

   p = new float [Xmax+1];     /* Dynamic Memory Allocation */
   pold = new float [Xmax+1];
   ChkArr = new int [Xmax+1];

   BigSum = TempSum = 0;
   Stop = I = 0;

   for(i=0; i <= Xmax; i++) {
    p[i] = pold[i] = 1;                        /* Begin Successive Relaxation Solution */
   }

  while ( !Stop ) {

    I++;                                       /* Next Iteration */
    if (I%10000 == 0) printf(".");

    for(i=0; i <= Xmax; i++) {
       pold[i] = p[i];                             /* Remember previous iteration value   */
       TempSum = 0;                                /* and reset temporary sum var */
       for(j=0; j<=Xmax ; j++) {
              TempSum += Q[j][i] * p[j];           /* Multiplying the matrices, p[i] modified */
       }
       p[i] = TempSum;
    /* printf("I = %d ; p[%d] = %f\n", I, i, p[i]);   Debug */
```

```
  }

  /* We do the testing ONLY at end of iterations */

  BigSum = 0;                                   /* UpDating BigSum after EACH new Iteration, I */
  for(i=0; i <=Xmax; i++)
    BigSum += p[i];

  for(i=0; i<=Xmax; i++) {
   if ( ((p[i] - pold[i])/BigSum < E) && ((p[i] - pold[i])/BigSum > -E) ) {
            ChkArr[i] = TRUE;
   }
   else ChkArr[i] = FALSE;
  }

 Stop = ChkArr[0];
 for(i=0; i<=Xmax; i++)                         /* If all old and current p's very close, finish */
  Stop = Stop * ChkArr[i];
}

 for(i=0; i<=Xmax; i++) {
  p[i] = p[i]/BigSum;
  /*printf("Final p[%d] = %f\n", i, p[i]);*/
 }
 record_results();                              /* write values to data file */
}
```

10. BIBLIOGRAPHY

[BECK88] R. Beck and H. Panzer, "Strategies for Handover and Dynamic Channel Allocation in Microcellular Mobile Radio Systems", *Proceedings 38th IEEE Vehicular Technology Conference*, pp. 178-184, June, 1988.

[CALL95] F. Callegati et al., "Call Admission Control for PRMA-based Multiservice Cellular Networks", *Proceedings of Australian Telecommunication Networks and Applications Conference*, pp. 485-489, Sydney, December 1995.

[COFF85] E. G. Coffman, T. T. Kadota and L. A. Shepp "A Stochastic Model of Fragmentation in Dynamic Storage Allocation", *SIAM J. Comput.*, vol. 14, no. 2, May 1985.

[COOP81] R. B. Cooper, *Introduction to Queueing Theory, Second Edition*, North Holland, New York, 1981.

[COX73] D. C. Cox and D. O. Reudink, "Increasing Channel Occupancy in Large Scale Mobile Radio Systems: Dynamic Channel REassignment", *IEEE Trans. Vehic. Tech.*, vol.VT-22, pp. 218-222, 1973.

[ETSI92] ETSI GSM Specifications, Series 01-12.

[GUER91] R. Guerin, H. Ahmady, M. Naghshineh, "Equivalent Capacity and its Application to Bandwidth Allocation in High Speed Networks", *IEEE J.S.A.C.*, vol. 9, pp. 969-981, 1991.

[HODG90] M. R. L. Hodges, "The GSM Radio Interface", *Br. Telecom Tech. J.*, vol. 8, pp. 31-43, January 1990.

[HUEB93] F. Huebner and M. Ritter, "Call and Burst Blocking in Multi-Service Broadband Systems with CBR and VBR Input Traffic", *Proceedings of MMB '93*, Aachen, Germany, pp.212-225, 1993.

[HUI89] J. Hui, "Resource Allocation for Broadband Networks", *IEEE J.S.A.C.*, vol. 6, pp. 1598-1608, 1989.

[KAUF81] J. S. Kaufman, "Blocking in a Shared Resource Environment", *IEEE Transactions on Communications*, vol. 29, no. 10, pp. 1474-1481, 1981.

[KELL91] F. P. Kelly, "Effective Bandwidth at Multi Class Queues", *Queuing Systems*, vol. 9, pp. 5-16, 1991.

[KLEI75] L. Kleinrock, *Queueing Systems Volume I: Theory*, John Wiley & Sons, New York, 1975.

[KNUT73] D.E. Knuth, *The Art of Computer Programming, Volume 2: Computer Algorithms*, Addison Wesley, New York, 1973.

[KONH86] A. G. Konheim and M. Reiser, "The moveable boundary multiplexor stability and decomposability," in *Teletraffic Analysis and Computer Performance Evaluation*. North Holland, New York, 1986.

[KWON95] D. Kwon and K. W. Sarkies, "System Performance Enhancement Combining Handover Rejection Scheme with Channel Reservation Scheme", *Proceedings of Australian Telecommunication Networks and Applications Conference*, pp. 497-502, Sydney, December 1995.

[LARS69] H. J. Larson, *Introduction to Probability Theory and Statistical Inference*, Wiley, New York, 1969.

[MONT95] R. Montagna, "TCH-HS Activities for the GSM Channel Standardisation", *Proceedings of ICC '95*, San Francisco, February 1995.

[MOUL92] M. Mouly and M. Pautet, *The GSM System for Mobile Communications*, Michel Mouly and Marie-Bernadette Pautet, France, 1992.

[PADG95] J. E. Padgett, C. G. Guenther, T. Hattori, "Overview of Wireless Personal Communications", *IEEE Communications Magazine*, vol. 33, no. 1, January 1995.

[RITT94] M. Ritter and P. Tran-Gia, *Multi-Rate models for Dimensioning and Performance Evaluation of ATM Networks*, Interim Report, COST 242 Project, Institute of Computer Science, University of Wuerzburg, 1994.

[ROBE81] J. W. Roberts, "A Service System with Heterogenous User Requirements - Application to Multi-Service Telecommunications Systems", *Proceedings of Performance of Data Communication Systems and their Applications*, G. Pujolle (ed.), North Holland, pp. 423-431, 1981.

[SIVA90] K. N. Sivarajan, R. J. McEliece and J. W. Ketchum, "Dynamic Channel Assignment in Cellular Radio", *Proceedings 40th IEEE Vehicular Technology Conference*, pp. 631-637, June, 1990.

[TELE78] Telecom Australia, *A Course in Teletraffic Engineering*, Prepared by the Staff of Traffic Engineering Section, Planning Services Branch, Headquarters, Melbourne, 1978.

[TRAN93] Tran-Gia P. and Huebner F., "An Analysis of Trunk Reservation and Grade of Service Balancing Mechanisms in Multiservice Broadband Networks", *IFIP Workshop TC6*, La Martinique, January 1993.

[USAI95] P. Usai, G. Cosier, D. Pascal, J. Sotscheck and M. Kappelan, "Subjective Performance Evaluation of the GSM Half-Rate Coding Algorithm (With Voice Signals)", *Proceedings of ICC '95*, San Francisco, February 1995.

[ZUKE88] M. Zukerman, "Circuit allocation and overload control in a hybrid switching system," *Comput. Networks ISDN Syst.*, vol. 16, pp. 281-298, 1988/1989.

[ZUKE89] M. Zukerman, "Bandwidth Allocation for Bursty Isochronous Traffic in a Hybrid Switching System", *IEEE Trans. on Communications*, vol. 37, no. 12, December 1989.