A NETWORK EMULATOR TO SUPPORT THE DEVELOPMENT OF ADAPTIVE APPLICATIONS

Nigel Davies, Gordon S. Blair, Keith Cheverst and Adrian Friday

Distributed Multimedia Research Group,
Department of Computing,
Lancaster University,
Bailrigg,
Lancaster,
LA1 4YR,
U.K.

telephone: +44 (0)524 65201
e-mail: nigel, gordon, kc, adrian@comp.lancs.ac.uk

Abstract

Mobile applications must operate in environments in which the network connectivity, input/output devices, power and contextual information available to them may all vary. Applications which react to changes in these parameters in order to ensure continuing service to the user are termed *adaptive applications* and have recently emerged as an area of intense research activity. In this paper we describe the design and implementation of a network emulator which facilitates research in this field by allowing applications to be exposed to user controlled fluctuations in network service. The emulator can be used with any application which uses UDP and requires only minimal changes to the application or, it may be used with applications written using the ANSAware distributed systems platform in which case no changes are necessary to the application. The design and implementation of the emulator are described in this paper as our experiences of using the emulator to model three distinct types of wireless network: GSM, an analogue cellular service and a simple shared radio channel. The source code for the emulator is freely available and instructions on obtaining the code are also included.

1. Introduction

Mobile computing environments are characterised by *variation*. In particular, during the execution-time of a mobile application the network connectivity, input/output devices, power and contextual information available to the application may all vary [Davies,94], [Duchamp,92], [Schilit,94]. In our research at Lancaster we are interested in developing applications and system services which are able to cope with wide fluctuations in these parameters (termed adaptive applications [Katz,94]) and in particular with fluctuations in the first of these parameters, i.e. network connectivity. This work has been motivated by two beliefs: firstly, that fluctuations in network connectivity are unavoidable in a mobile environment and secondly, that such fluctuations should become an accepted part of an application's operation and not be treated as an error or temporary 'glitch'.

The first of these statements is clearly true if we assume that mobile computers will have multiple network interfaces [Hager,93] and that users will be able to dynamically switch between networks (for example switching between a wired network and a local area wireless network when un-docking a portable PC). Determining the feasibility of the second of these statements requires further research. However, to conduct this research requires an environment in which the level of network service available to a mobile machine can be varied. This paper reports on the development of such an environment based not on hardware but on a software network emulator which allows us to conduct research into network variance without investing in multiple network infrastructures.

Section 2 of this paper describes the overall design and configuration of the network emulator. Details are given of how the emulator may be configured to emulate a number of different networks with examples based on three wide-area wireless networks with which we have practical experience: GSM, a U.K. analogue cellular service and an analogue private mobile radio (PMR) system. Section 3 then briefly describes a graphical user interface to the emulator which enables users to control and visualise the flow of information between mobile computers. Section 4 presents details of the modifications necessary to client applications to enable them to exploit the emulator. Particular emphasis is placed on the use of applications written using the ANSAware distributed systems platform which we have modified to operate with the emulator. Section 5 then presents an analysis of the performance of the emulator and highlights the relationship between the network bandwidth to be emulated and the average size of packets sent to the emulator. Finally, section 6 contains some concluding remarks.

2. Design and Implementation

2.1. Emulator Design

The network emulator is designed to provide an approximate emulation of low-speed networks using standard hardware and systems software. It should be stressed that as researchers we are more concerned with variance in network connectivity than precise simulations of network characteristics and hence the accuracy of the emulator is not considered to be of critical importance (for example, we use standard UNIX timing facilities). The basic approach used by the emulator is to intercept UDP packets travelling between sources and sinks and to introduce a delay similar to that which would be incurred if the packets were transmitted over a slower network. The emulator mimics the workings of a slow speed network and so delays are related to (for example) network load and error rates. The most controversial feature of the emulator's design is that it is structured as a single, central point through which all messages are routed and at which point network delays are introduced. Thus for each node in the network to be emulated the emulator maintains a queue of packets waiting to be transmitted. This use of a central point for the emulation is in contrast to systems such as Ingham's Delayline network emulator [Ingham,94] in which processing is carried out at both the sender and recipient of messages with delays being implemented at the receiver's end.

While implementing the emulator as a central point clearly creates a bottleneck in the system there are two key advantages to be gained from this approach. Firstly, the emulator is able to adjust the network characteristics experienced by applications based on load. Hence, for example, if the network to be emulated has a simple shared transmission medium the emulator itself can detect potential packet collisions and discard the appropriate packets. The second advantage is that the semantics of sockets are automatically preserved by the emulator: the sender always believes that packets have been sent properly since they always appear to reach their destination (in practice of course they have only reached the emulator) and the receiver receives messages in the order in which they would arrive in a real network (in contrast to the Delayline system in which packets may arrive in the wrong order since the delay is introduced at the receiver side of the communication).

However, there are a number of disadvantages in structuring the network emulator as a central process. In particular, the design makes the following assumptions:-

- The time the emulator takes to process a packet is negligible compared to the delay incurred during transmission over a slow network.
- The time taken to transmit a UDP packet over the high speed network is negligible compared to the delay incurred during transmission over a slow network.
- The number of nodes that are to be interconnected via the emulator is small (i.e. less than sixteen) and only a subset of these are transmitting at any one time.

Clearly, as the size of UDP packets decrease or the speed of the network being emulated increases then the first two of these assumptions introduce increasing inaccuracies. However, in practice we have found that these assumptions are valid for the type of experimental work we wish to carry out (see section 5).

2.2. Emulator Configuration

The network emulator can be configured in two distinct ways. Firstly, new types of network may be introduced, e.g. a connection oriented cellular service or a connectionless shared medium network. This requires modification to one of the emulator's source files and re-compilation. In more detail, the user must supply a function called new_network_name_send (senderNodeId, dataPacket) which is called by the emulator every time a packet is to be sent via the new network. Within this function the user must implement any delays which are associated with attempting to send packets on the network. For example, if the network has a high turn-around time which occurs when the node switches from receiving to transmitting information this can be modelled with the new_network_name_send function. Error characteristics can also be specified for the network or the occurrence of errors may be modelled as part of the throughput specified. Once a new network has been introduced its behaviour can be tailored during run-time using configuration files. A typical configuration file is shown in figure 1.

Line 1 of the configuration file denotes the type of the network to emulate - in this case a raw radio channel. Line 2 specifies the number of nodes that are connected to the network (in this case 3) and lines 3 to 5 provide information about each node. Specifically, for each node its name, maximum buffer size and internet address must be specified. The buffer size is used to

1	raw	<network to be emulated>
2	3	<no. of sources/sinks of data>
3	0 1044000 148 88 16 27 columbine	<source no. 0>
4	1 1044000 148 88 16 25 sinbad	< source no 1>
5	2 1044000 148 88 32 2 edc2	<source no. 2>
6	0 0 1 1200 3c	< channel characteristics 0-1>
7	1 0 2 1200 3c	< channel characteristics 0-2>
8	2 1 0 1200 33	< channel characteristics 1-0>
9	3 1 2 1200 33	< channel characteristics 1-2>
10	4 2 0 1200 0f	< channel characteristics 2-0>
11	5 2 1 1200 0f	< channel characteristics 2-1>

Figure 1 : A Typical Configuration File

prevent applications from running ahead of the network; once a node's buffer is full all subsequent send requests will be blocked.

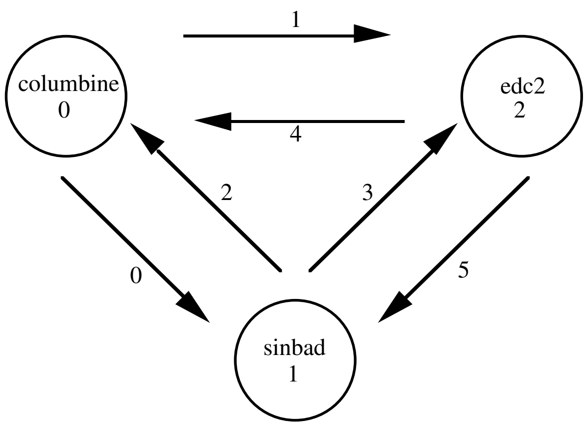

Figure 2 : Emulator Channel Connection Diagram

For the purposes of the emulator the network to be emulated must be visualised as a series of uni-directional channels interconnecting each of the nodes (see figure 2). The characteristics of these channels are specified in lines 6-11. For example, line 6 of the configuration file specifies the characteristics of channel 0, i.e. the channel between nodes 0 and 1. The characteristics are that the channel has a throughput of 1200 bps and that messages transmitted on this channel collide with messages transmitted simultaneously on any of the *other* channels except channel 1, i.e. the other outgoing channel from this node. This is expressed using a bit map mask with a bit being set denoting that collisions occur with messages on the corresponding channel (channel 0 being the least significant bit).

The network emulator supports dynamic updates to the configuration file during operation so that the effect of changing the quality-of-service of a network can be easily demonstrated. For example, by simply setting the throughput of a given node's output channels to zero we can emulate disconnection. It should be noted however that radical changes to the network configuration may result in those packets currently being queued at the emulator being delayed longer than expected during the reconfiguration. This is because when the emulator is re-configured it re-calculates the dispatch time of all the waiting packets without taking into account any time the packets have already been delayed. Thus re-configurations involving small changes to the throughput of slow-speed networks are most susceptible to this problem (particularly if the packets being queued are relatively large). Re-configurations involving the addition or removal of nodes are supported: the emulator simply prints a warning if a previously supported node is no-longer supported in the new configuration file.

2.3. Example Configurations

We have used the emulator to emulate three types of network: GSM, a U.K. analogue cellular service and a simple shared radio channel. In the case of GSM the network appears to have fairly dependable characteristics with an average call set up time of 3 seconds and a corrected throughput of 9600 bits/sec. In the case of the analogue cellular service (using Motorola Cellect modems and MicroTac II handsets) the call set up time is substantially longer, taking about 20 seconds for the connection to be fully established. Once the connection has been established we are able to get a corrected throughput of around 3700 bits/sec on a theoretical 4800 link. Setting the modems to run at higher speeds typically gives us a lower corrected throughput due to the number of retransmissions necessary to compensate for the high error rate.

The emulator configuration for these two networks is very similar. Two new send routines were required (gsm_send () and analogue_send ()) which introduced the appropriate connection delay for each network. For both networks packets sent by a node to a new destination causes the emulator to simulate the disconnection of the node from its previous destination and connection to the new destination node. The only difference between the gsm_send and analogue_send routines is the length of delay they introduce to emulate call connection. The configuration files for these networks are both straightforward with the collisions flags being set to no collisions and the throughput being set at 9600 and 3700 for the GSM and analogue networks respectively.

```
raw_send ( int : sourceId, dataPacket *pkt)
{
    mapAddressToNodeId (dataPkt->
                destinationAddress, &sinkId);
    if (emptyQ(sourceId) {
        addToTxQ (sourceId, pkt);
        obtainChannelCharacteristics(sourceId,
                sinkId, &characteristics);
        /* insert any additional delays/errors here */
        calculateDispatchTime(pkt->length,
        characteristics->bandwidth);
        resetCollisionBits (characteristics);
    }
    else
        addToTxQ (sourceId, pkt);
}
```

Figure 3 : Pseudo Code for Emulator Send Routine for a Raw Radio Channel

The emulation of the raw radio channel has a much more straightforward send routine which introduces no additional delays (see figure 3). However, the configuration file for this type of network is more complex. In particular, the collision flags must be set such that data on any channel collides with data on any other channel. For the purposes of our work we have used a throughput of 1200 bits/sec to emulate the characteristics of a simple analogue private mobile radio (PMR) system. An example configuration file for this type of network is given in figure 2.

3. A Graphical Interface to the Network Emulator

At an early stage in the network emulator's development it was realised that a graphical front-end to the emulator could be used to enhance demonstrations of adaptive applications. The interface we have developed allows users to both view and control the operation of the emulator. During normal use the interface displays for each node the number of packets waiting to be dispatched, the last action that occurred with respect to that node (e.g. packet arrived, packet dispatched etc.) and for the packet at the head of the node's queue its destination, size and dispatch time. Hence, if we have a fast sender connected to its intended destination by a slow network the queue size for the sending node will build up steadily and we will see many more packet arrival events than packet departures. The interface to the emulator also allows users to control the emulator by dynamically changing the configuration file it uses. In this way we can, for example, show the effect on applications of gradually reducing the throughput available.

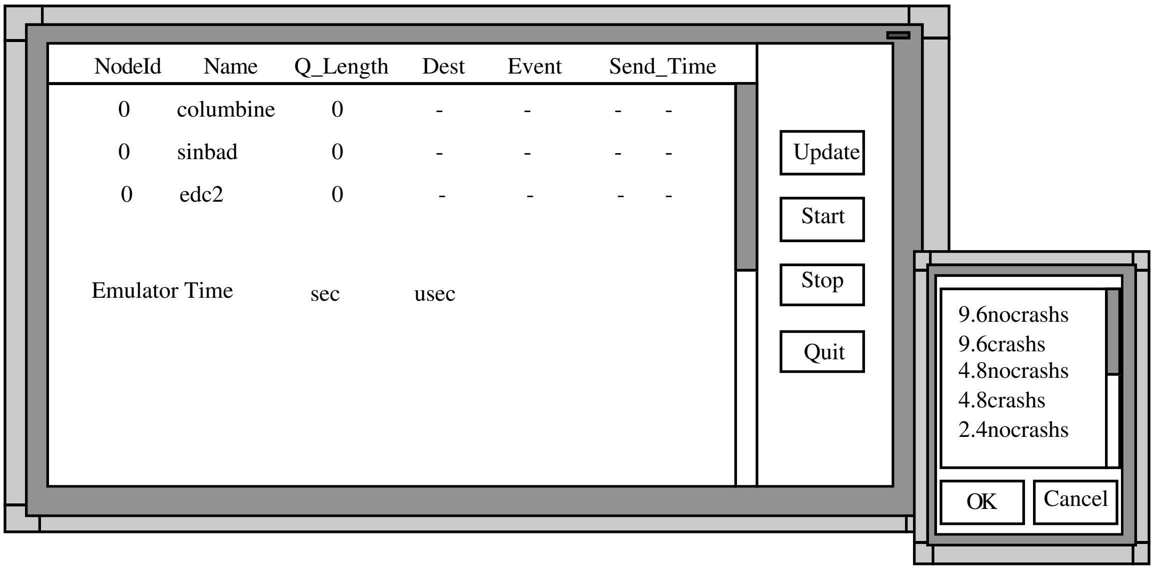

Figure 4 : The Network Emulator Controller

The interface is implemented as an entirely separate process which communicates with the emulator using sockets. This communication takes the form of well-defined packets sent from the emulator whenever a relevant event occurs (see figure 5). Control packets to, for example, force the emulator to update its configuration file can be sent from the monitor to the emulator.

```
struct monitorPacket {
    char header [2];      /* identifies emulator pkt */
    int type;             /* type of pkt */
    struct timeval
        time_now;         /* emulator's clock time */
    int nodeId;           /* node to which msg relates */
    int qLength;          /* length of q for this node */
    int event;            /* event which has occurred */
    int size;             /* size of data pkt involved */
    int dest;             /* destination nodeId */
    struct timeval
        event_time;       /* time relating to the event.
                             Can be different for each
                             event */
};

/* pkt has been discarded */
#define EVENT_PKT_DISCARD       0
/* pkt has been sent */
#define EVENT_PKT_SENT          1
/* pkt collided (and discarded) */
#define EVENT_PKT_COLLIDED      2
/* pkt has arrived at node */
#define EVENT_PKT_ARRIVED       3
/* pkt has been scheduled for tx */
#define EVENT_PKT_SCHEDULED     4
```

Figure 5 : Packet Format For Communications Between the Emulator and its Interface

Implementing the emulator's user interface as a separate process has the two distinct advantages. Firstly, we can run the interface on a separate machine and thus could implement processor intensive graphics monitoring tools without affecting the performance of the emulator. Secondly, we can have a number of different interfaces implemented to illustrate and control different aspects of the emulator.

4. Emulator Client Code

4.1. Standard Distributed Applications

We use the emulator with two types of distributed application. The first are standard distributed applications which communicate using UDP. In order that these can use the emulator they must use new versions of the sendto and recvfrom system calls. These are currently implemented as new functions emulator_sendto and emulator_recvfrom which form wrappers around the standard calls in order to add and remove additional header information required by the emulator. Applications must at present be re-compiled to use these new functions. However, it would be a relatively straightforward task to compile these functions as a library which could be dynamically linked with existing applications to allow them to transparently use the emulator. The format of the packet headers used by emulator_sendto and emulator_recvfrom is shown in figure 6.

```
struct dataPacket {
  char    header [2];               /* identifies em. pkt */
  int     type;                     /* type of pkt */
  struct  sockaddr_in toAddr;       /* destination */
  struct  sockaddr_in frAddr;       /* source address */
  struct  sockaddr_in ackAddr;      /* address to ack.
                                         transmission */
  int     bufLen;                   /* length of user data */
  char    *buf;                     /* user data */
};
```

Figure 6 : Structure of an Emulator Data Packet

The header field identifies the packet as being associated with the emulator. It is used by the emulator to check that it is receiving valid packets and by the emulator_recvfrom function to determine whether or not to strip off the header before passing the buffer up to the application. The type field is used to distinguish between data and control packets. Data packets are those which are passed to the emulator for subsequent dispatch to a destination mode. Control packets are used to control the emulator's behaviour and typically originate from the emulator's user interface. In addition data packets can be flagged as those requiring an acknowledgement that the packet has been queued for transmission, those that require an acknowledgement of transmission and those which require no acknowledgement at all. This allows us to implement synchronous emulator_sendto routines for those applications which would otherwise 'run-away' or cause congestion when operating over a low-speed link. If the packet requires an acknowledgement the emulator sends this to the address specified in the ackAddr field.

The first two address fields are used by the emulator to ensure that the packet is transmitted to the appropriate final destination and, at the destination, to ensure that the application believes that the packet originates from the initial source rather than from the emulator. The ack_address is used for flow control between the emulator and the source application as described above.

4.2. ANSAware Applications

The second type of application we have used with the emulator are those based on the ANSAware distributed systems platform [APM,89]. This software suite is itself based on the ANSA architecture which has had a profound influence on the RM-ODP [ISO,92]. Thus, the platform tackles the problem of developing applications to operate in a heterogeneous environment. The ANSA programming model is based on a location-independent object model where all interacting entities are treated uniformly as encapsulated objects. Objects are accessed through operational interfaces which define named operations together with constraints on their invocation. Objects are made available for access by exporting interfaces to a special object known as the *trader*. An object wishing to interact with this interface must then import the interface from the trader by specifying a set of requirements in terms of a interface type and attribute values. This will be matched against the available services and a suitable candidate selected. At this stage, an implicit *binding* is created to the object supporting the interface, i.e. a communication path is established to the object. Invocation of operations can then proceed.

To provide a platform conformant with the above programming model the ANSAware suite augments a general purpose programming language (usually C) with two additional languages. The first of these is IDL (Interface Definition Language), which allows interfaces to be precisely defined in terms of operations, arguments and results. The second language, DPL (Distributed Processing Language) is embedded in a host language, such as C, and allows interactions to be specified between programs which implement the behaviour defined by these interfaces. Specifically, DPL statements allow the programmer to import and export interfaces, and to invoke operations in those interfaces (see figure 7).

! {stack} <- traderRef$Import ("Stack", "*context*", "*properties*")

! {result}<-stack$Push (value)

Figure 7 : Example DPL Statements

In the engineering infrastructure, the binding necessary for invocations is provided by a remote procedure call protocol known as REX (Remote EXecution protocol) or a group execution protocol know as GEX (Group EXecution Protocol). These are layered on top of a generic transport layer interface known as a *message passing service* (MPS). A number of additional protocols may be included at both the MPS and the execution protocol levels and these may be combined in a number of different configurations. The infrastructure also supports lightweight threads within objects so that multiple concurrent invocations can be dealt with.

All the above engineering functionality is collected into a single library, and an instance of this library is linked with application code to form a *capsule*. Each capsule may implement one or more computational objects. In the UNIX operating system, a capsule corresponds to a single UNIX process. Computational objects always communicate via invocation at the conceptual level but, as may be expected, invocation between objects in the same capsule is actually implemented by straightforward procedure calls rather than by execution protocols.

We have developed a modified version of the ANSAware libraries which includes code to route packets generated as a result of object invocations via the emulator. By use of a single function call the application can optionally enable one or other of the synchronous transmission modes supported by the emulator, i.e. application is blocked until messages are queued or application is blocked until messages are transmitted. Running ANSAware applications over the emulator highlighted a number of shortcomings in the ANSAware remote procedure call protocol REX. More specifically, REX is tuned to run on a moderately loaded Ethernet and does not implement any form of congestion control. In addition, the tuning parameters are specified at compile time which makes it impossible for REX to adapt to changes in network bandwidth.

We have implemented a new remote procedure call package for ANSAware called QEX (Quality-of-service remote EXecution protocol). QEX differs from REX in that it is specifically designed to operate over a wide range of network types adapting seamlessly to changes in network quality-of-service. This is achieved by analysing sequences of messages to determine the round-trip time between client and server. These round-trip times are smoothed to eliminate network jitter (processing at the server end ensures that application delays are eliminated from the calculation) and then form the basis of tuning parameters. In particular, retry rates are calculated to avoid unnecessary network congestion while ensuring that packet losses are detected as early as possible. Quality-of-service information is maintained on a per-session basis and hence the protocol is able to accommodate simultaneous object interactions over differing networks (e.g. if a client is talking to two services one of which is located on a mobile host while the other is on a high-bandwidth fixed network).

In addition to using quality-of-service information for tuning purposes QEX is also able to provide feedback to applications on the state of the underlying communications channels. To facilitate this we have introduced the notion of explicit bindings into the ANSAware platform. Explicit bindings are established using a bind operation which takes as parameters the source and sink interfaces to be bound and a further set of parameters which express the desired quality-of-service. Clients are returned a binding control interface as a result of the bind operation through which they can register for call-backs if the specified quality-of-service is violated. These call-backs are generated by QEX based on the information it collects for tuning purposes and allow applications to adapt to changes in the network characteristics. In this way applications can provide feedback to users on the state of the network and congestion control strategies can be adopted by applications and users in addition to the underlying protocol.

QEX has been largely developed using the network emulator which has allowed us to simulate rapid fluctuations in network quality-of-service and thus refine our algorithm for calculating retry rates. More details on QEX can be found in [Davies,94].

5. Performance

We have tested the accuracy of the network emulator over a range of different network speeds and with varying numbers of clients transmitting different packet sizes. The graphs in this section can be used to ascertain the optimum configuration file settings for a given combination of network speed and average packet size. All of the figures were taken using a network of Sun Sparx1 machines running SunOS 4.1 and interconnected using Ethernet. The emulator ran on a separate machine to the clients and servers and all the machines and the network were 'lightly loaded' at the time of testing.

To obtain the figures we ran simple client/server pairs in which the client repeatedly sent fixed size buffers to the server. The server recorded the time taken to receive a set number of these buffers and from this timing information calculated the average throughput. Standard Unix timing facilities were used throughout.

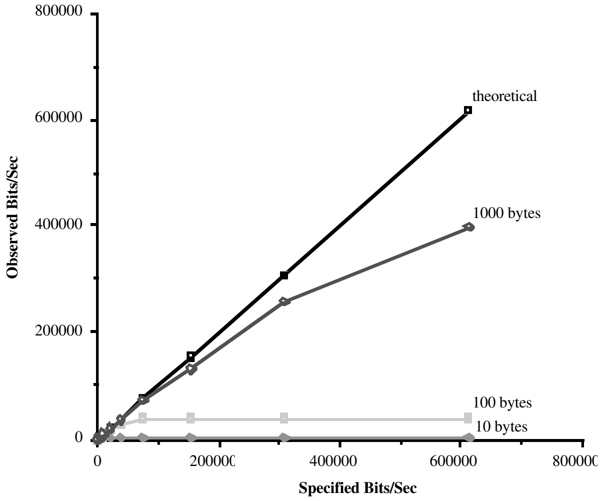

Figure 8 : Network Emulator Performance For Single Client/Server Pair in the Range 0-614400

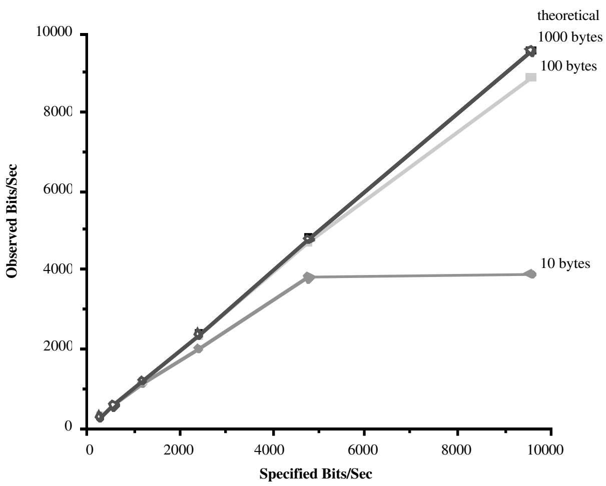

Figure 9 : Network Emulator Performance For Single Client/Server Pair in the Range 0-9600

Figures 8 and 9 shows the emulator's performance for a single client/server pair of processes. Figure 9 is based on the same timings as figure 8 but the graph shows a narrower range of network bandwidths in order to improve the level of detail which can be observed. In both graphs the x-axis is the bandwidth as specified in the configuration file and the y-axis is the observed bandwidth. The different lines denote different packet sizes (10, 100 and 1000 bytes).

The key thing to note from these graphs is that the accuracy with which the emulator models the network bandwidth is heavily dependent on the packet size. Moreover, for any given packet size there is a maximum speed at which the emulator can process and dispatch the packets. Increasing the bandwidth in the configuration file has no effect on the observed bandwidth above this cut-off point. In our tests the cut-off points were as follows: the maximum observable throughput with 10 byte packets was 3998 bytes; the maximum observable throughput with 100 byte packets was 39978 bytes and the maximum observable throughput with 1000 byte packets was 399792 bytes.

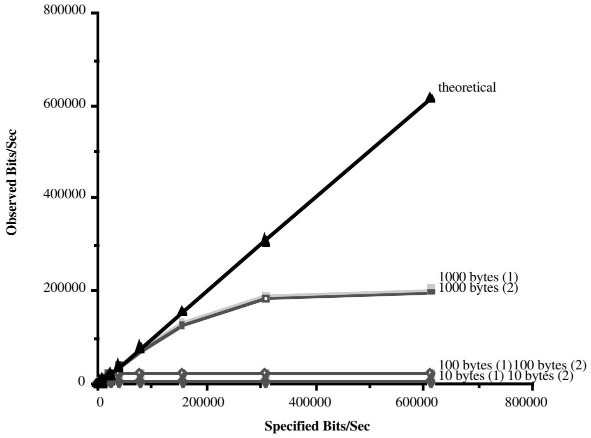

Figure 10 : Network Emulator Performance For Two Client/Server Pairs in the Range 0-614400

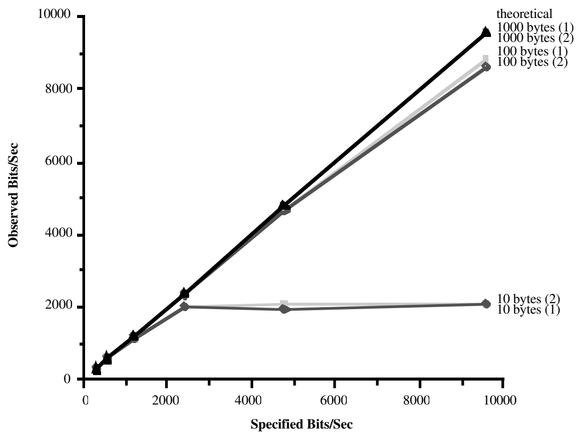

Figure 11 : Network Emulator Performance For Two Client/Server Pairs in the Range 0-9600

Figures 10 and 11 illustrate how the performance of the emulator degrades with the addition of a new client-server pair. For these figures the emulator was driven by two clients, both sending fixed size packets at their maximum rate. The graphs show the two different traces (one for each client) for the same packet sizes as above. Once again the cut-off points are evident with the maximum observable throughput with 10 byte packets being 2439 bytes; the maximum observable throughput with 100 byte packets being 19985 bytes and the maximum observable throughput with 1000 byte packets being 201000 bytes. As might be expected while the addition of new client/server pairs impacts on the performance of the emulator this impact is evenly distributed between the client/server pairs such that both see an almost identical (though less accurate) throughput.

The frequency with which the situation depicted in figures 10 and 11 occurs is clearly application dependent. In our work at Lancaster we have been focusing on the development of collaborative mobile applications for use by field engineers in the utilities industries. As a result, we have been mainly interested in emulating the type of low-speed radio networks suitable for wide-area use. In addition, the collaborative applications we have written typically have a fairly well-defined request-reply style interaction based on packet sizes of around 100 bytes and as a result we typically do not have multiple processes transmitting large numbers of messages concurrently. For this type of application the emulator has proved more than adequate and enabled us to make substantial progress in application development prior to obtaining wide-area mobile communications hardware. For more demanding applications with multiple nodes transmitting concurrently the emulator's performance can be improved by replication. In the degenerate case a separate network emulator can be used for each source node. In this case however, the emulator is only able to provide functionality equivalent to that found in Delayline since there is currently no mechanism defined for separate instances of the emulator to communicate in order to support packet collisions etc. Experimentation would be required to determine if such a distributed co-ordination protocol could be implemented while still allowing the emulators to function at level significantly better than a centralised version.

6. Concluding Remarks

This paper has described a network emulator developed at Lancaster to enable research into adaptive applications. It should be stressed that the system described provides an *emulation* of low-speed networks *not a simulation*, i.e. real applications can be compiled and executed using the emulator and these applications will experience a level of network service similar to that which they would experience if they were running over real low-speed networks. The design and implementation of the emulator has been described as has the design and implementation of a separate graphical front-end and monitoring tool for the emulator.

The performance of the emulator has been evaluated and those applications for which the emulator is best suited identified. In particular, the impact of small message sizes on the emulator's accuracy has been discussed.

The emulator and its front end have been successfully compiled and run on SUN Sparcs running SunOS, SUN Sparcs running a soft real-time version of SunOS 4.1 [Hagsand,94] and portable 486 PCs running SVR4. Sources for the emulator and the front-end are available via anonymous ftp from ftp.comp.lancs.ac.uk. In addition, the URL:

http://www.comp.lancs.ac.uk
/computing/users/nigel/emulator.html

provides more information on the network emulator and access to the source code for both the emulator and the interface described in this paper.

Acknowledgements

The work described in this paper was initiated by one of the authors while a visiting researcher at the Swedish Institute of Computer Science. The authors would like to acknowledge the contribution of Olof Hagsand who wrote the code to enable the emulator to use SICS's soft real-time SunOS and Steve Pink for discussions during the development of this emulator.

References

[APM,89] A.P.M. Ltd. "The ANSA Reference Manual Release 01.00", APM Limited, UK. March 1989.

[Duchamp,92] Duchamp, D. "Issues in Wireless Mobile Computing." Proc. Third Workshop on Workstation Operating Systems, Key Biscayne, Florida, U.S., 1992. IEEE Computer Society Press, Pages 2-10.

[Davies,94] Davies, N., G.S. Blair., K. Cheverst and A. Frdiay "Supporting Adaptive Services in a Heterogeneous Mobile Environment." Proc. 1st International Workshop on Mobile Computing Systems and Applications, Santa Cruz, U.S., December 1994.

[Hager,93] Hager, R., A. Klemets, G.Q. Maguire, M.T. Smith, and F. Reichert. "MINT - A Mobile Internet Router." Proc. IEEE VTC'93, Secaucus, NJ, U.S., 1993.

[Hagsand,94] Hagsand, O., and P. Sjödin. "Workstation Support for Real-time Multimedia Communications." Proc. USENIX Winter 1994 Technical Conference, 1994.

[Ingham,94] Ingham, D. B. and G. D. Parrington "Delayline: A Wide-Area Network Emulation Tool." Technical Report, Department of Computer Science, University of Newcastle, Newcastle upon Tyne, NE1 7RU, U.K.

[ISO,92] ISO. "Draft Recommendation X.901: Basic Reference Model of Open Distributed Processing - Part1: Overview and Guide to Use", Draft Report 1992.

[Katz,94] Katz, R.H. "Adaptation and Mobility in Wireless Information Systems." IEEE Personal Communications Vol. 1 No. 1, Pages 6-17.

[Schilit,94] Schilit, B., N. Adams and R. Want "Context-Aware Computing Applications" Proc. 1st International Workshop on Mobile Computing Systems and Applications, Santa Cruz, U.S., December 1994.