Emulator Express: A system for optimizing emulator performance for wireless networks

by B. C. Housel
I. Shields

IBM eNetwork™ Emulator Express is an IBM program product that optimizes the operation of Telnet 3270 and 5250 emulation over extremely low-bandwidth networks. These optimizations enable mobile workers using laptops, notebooks, or other mobile devices to access legacy host applications effectively over wide-area wireless networks as well as low-bandwidth wireline modem connections. This paper describes how the Emulator Express system intercepts the data stream and optimizes it transparently to both the client emulator and the Telnet server. The optimizations include a new data stream caching technology, a new optimized protocol that reduces the number of Telnet negotiation flows, and traditional compression. The data stream caching technology is particularly significant because it may be applied to other distributed application domains. The results of several performance experiments are reported that illustrate the improvements in data transport volume and response time when using Emulator Express.

This paper describes the design of Emulator Express (EE), an IBM program product[1] that optimizes the operation of TN3270 and TN5250 emulation to enable mobile workers using laptops, notebooks, or other mobile devices to effectively access legacy host applications over wide-area wireless networks as well as low-bandwidth wireline modem connections. EE is part of IBM's SecureWay* and eNetwork* wireless product offerings, including the SecureWay Wireless Gateway and SecureWay Wireless Mobile Client software.[1] EE is middleware that can be used with any wireless or wireline technology that implements the Transmission Control Protocol/Internet Protocol (TCP/IP). As confirmation of its success, some customers using EE technology report that the performance of running host applications when connected to their wireless networks often exceeds that of running the same applications when connected to a local area network (LAN).

Emulation. The IBM 3270 and 5250 display terminals are fixed-function, buffered, block-mode terminals with a certain amount of formatting capability built into the display. A large amount of the world's data is accessible through such terminals, and they are common in many types of businesses. The original terminals used binary synchronous communication (BSC) and later Systems Network Architecture (SNA) protocols to communicate with a host. Many emulators of these terminals are now available for personal computers, and it is common for emulators to use TCP/IP and a specific Telnet regime (TN3270 or TN5250) to communicate to a host computer either directly or via a separate TN3270 or TN5250 server that in turn is attached to the host using SNA protocols.

0018-8670/00/$5.00 © 2000 IBM

The widespread use of today's emulation products testifies to the value of emulation. Although new Web or Internet technology is slowly replacing the legacy systems, access to these legacy systems will be important for the foreseeable future. Indeed, new emulators, written in the Java** programming language, such as IBM's Host-On-Demand,[2] are designed to access legacy systems.

Gateway mechanisms on various Web servers and gateway products contain functions to access host applications using SNA or Telnet emulation protocols; e.g., the Customer Information Control System (CICS*) Transaction Gateway[3] supports 3270 emulation to access host applications. Other recent developments in the emulation area have focused on generalized tools that enable customers to map 3270 screens to customized Web pages. These developments include IBM's SecureWay Host Publisher,[4] Host Access Class Library (HACL),[5] and the Internet Engineering Task Force (IETF) TN3270E working group draft on Open Host Interface Objects (OHIO).[6]

The need for emulation over wireless networks. Many businesses and public service entities such as police departments have mobile field forces or officers who are in radio contact with some form of base. Typically, such persons may be sent to a job or location by a dispatcher who uses a 3270 or 5250 terminal connected to a central computer system. Additional information from the same or a different system may also be retrieved for particular cases, such as a criminal record check associated with a license plate check when a police officer pulls over a driver of a vehicle for a traffic infraction.

In such cases the dispatchers may become a bottleneck, particularly in crisis situations. Furthermore, a dispatcher may not relay all available information or may make errors. Finally, the voice channel itself is not always clear. For such environments, a mobile computer in the field person's vehicle is a promising way of reducing delay and increasing accuracy of information delivered to the mobile user. Ideally, the field person given such a solution should be able to access the same information as the desk-bound dispatcher could access, preferably without extensive reprogramming of the underlying information systems.

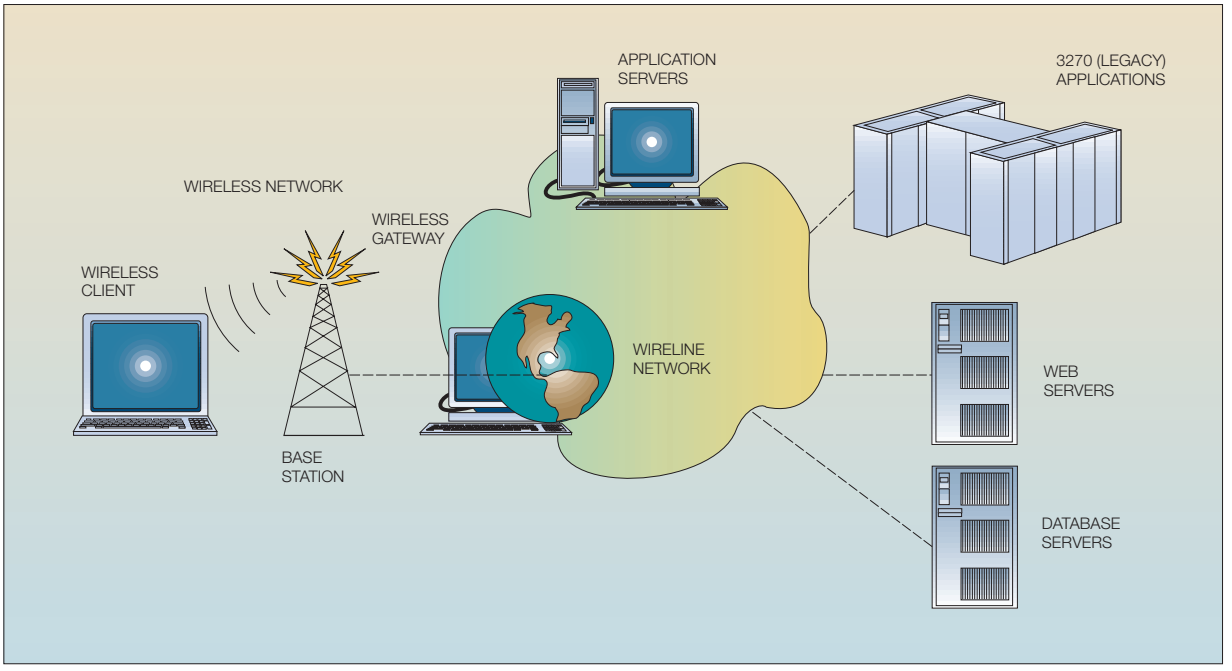
The promise of wireless technology. Wireless technology has not yet lived up to expectations such as those described in Imielinski and Badrinath.[7] There are many reasons for this, not least of which is the inherently low throughput over most commercially available packet radio networks. Although the nominal connection speed at the mobile client is frequently as high as 19200 bps, the radio transmission speed may be slower, and the nature of packet radio is such that the packets flow serially through several connections, further reducing the effective throughput for interactive applications. A typical environment is shown in Figure 1.

The widespread use of today's emulation products testifies to the value of emulation.

In addition to the low bandwidth, the communications channel is usually subject to high error rates because of radio interference, low power available for operating the radio components in mobile units, and poor or varying signal coverage. The high error rates often exacerbate recovery when a packet that may have actually reached the target destination is retransmitted by the sender because the acknowledgment did not arrive in time. Although the second transmission is unnecessary, it will delay any following packets while it is transmitted and possibly retransmitted by the underlying radio network equipment or drivers. This problem becomes more serious as packet sizes increase, resulting in much greater times both for the actual packet transmission and, even more importantly, for the time that the packet spends in queues waiting to be transmitted. In this environment, the probability of retransmission rises dramatically, and the retransmission attempts cause further channel degradation.

For TN3270 and TN5250 emulation to be effective over wireless networks, it is essential to significantly reduce the amount of data normally transferred between a host application and a 3270 or 5250 terminal emulator over such a TCP/IP connection. This reduction is particularly important in packet radio networks where a private communications protocol usually underlies the Internet Protocol (IP) transport. For example, the ARDIS network uses a packet size of 240 bytes and the RAM network a packet size of

Figure 1 Typical connection for a wireless mobile worker



512 bytes. Both of these networks limit the number of packets that may be outstanding without acknowledgment. For the purpose of illustration let us assume that the time required to transmit a radio packet or acknowledgment is a constant t. If a message fits into a single packet, it can be operated on by the receiver after time t. However, if two radio packets are required, it may take t for the first to arrive, t for the acknowledgment to be returned, and another t for the second packet to arrive. Adding even a few bytes to a message may thus triple the radio transfer time required before the receiver can use the message.

Traditional approaches. Several methods exist for achieving the goal of delivering 3270 or 5250 data to a mobile unit. The first, and possibly most obvious, is simply to compress the data stream. Compression can be done by the radio network drivers as is done by the IBM SecureWay Wireless Gateway and SecureWay Wireless Mobile Client software. The compression function is similar to the compression scheme used by land-line modems. Each packet is compressed independently of others. Compression tends to be more effective with more data stream history, so this method, although reasonably effective, is far from optimal.

Another approach to delivering data to mobile units is to implement the 3270 or 5250 emulator on the land-line side of the communications channel and use a program to *scrape* information from the screen, typically by using the Extended High-Level Language Application Programming Interface (EHLLAPI) available on many platforms to allow programmatic access to the display contents and formats. Such approaches may be specific to a particular application set and its screen formats, or they may be more general and combine more efficient compression than is possible for independent packets along with some form of screen caching. Either of these screen scraping approaches requires corresponding software on the mobile side to reconstitute the scraped data into a format for presentation to the mobile user, either as a literal image of the original or a customized abbreviated form. The latter method has been used by vendors such as Telxon, and the former method was embodied in the IBM ARTour server.[8,9]

One major drawback to the EHLLAPI screen scraping approach is that not all of the 3270 or 5250 data stream is available to the EHLLAPI application. This drawback is particularly important for 5250 applications that use advanced field characteristics, such as auto-fill or auto-entry, which are not available to

EHLLAPI applications. Another major deficiency is the requirement to have, for general emulation, a special emulator to rebuild the display on the client device. Thus, users cannot use their emulator of choice. This approach is suited to an environment where specific application programming is done for the mobile unit to adapt existing applications for mobile use. It is not well-suited to a general approach that attempts to reconstitute arbitrary application screens at the mobile unit, because it cannot do so with fidelity on account of the EHLLAPI limitations.

Emulator Express approach. The IBM Emulator Express product provides efficient emulation over low-bandwidth network connections without the limitations described above. EE provides dramatic improvement over wireless networks and also noticeably improves emulator performance over modem land-line connections. More specifically, the main objectives of the EE product are to:

- Optimize the delivery of 3270 and 5250 data to mobile users over TCP/IP connections using Telnet (TN3270 and TN5250) protocols, thus enabling mobile clients to use 3270 or 5250 emulation over low-bandwidth (e.g., wireless) connections with acceptable performance in terms of cost and response time. The use of TCP/IP-based transport is consistent with the directions taken by wireless network providers: the Cellular Digital Packet Data (CDPD) protocol supports IP; the IBM SecureWay Wireless Gateway and SecureWay Wireless Mobile Client provide an IP interface over a public packet radio network such as ARDIS and RAM; GSM (Global System for Mobile Communications) provides TCP/IP transport over public switched networks.
- Operate with any vendor's emulator or Telnet server. Several vendors manufacture 3270 or 5250 emulators that operate on a personal computer and use TN3270 or TN5250 forms of the Telnet protocol over TCP/IP networks to connect to host computers. Most of these work with EE so customers are usually able to continue using familiar emulators.
- Operate without imposing any restrictions on the data delivered to the emulator over the Telnet protocol.
- Be easy to install and configure. Ease of use was considered to be very important for gaining market acceptance. EE requires a minimum of additional configuring, and client configurations can be updated dynamically at run time.

Unlike screen-scraping techniques that operate on screen display buffers, the EE data reduction algorithms operate on the native 3270 or 5250 data stream directly. EE employs three optimization techniques: caching of data stream segments, compression of transmission buffers, and Telnet protocol reduction. Caching and compression are aimed at significantly reducing the volume of data transmitted over the network, whereas Telnet protocol reduction reduces the number of flows during session establishment to improve initial connection time and also eliminates problems associated with sending keep-alive messages to out-of-range mobile units. A detailed discussion of each of these techniques comprises the rest of the paper.

The next section gives a general overview of the Emulator Express system. The third section describes the additional configuration required for EE. The fourth section discusses the TCP/IP and Telnet communications aspects of EE and other optimizations that were implemented to improve initial connection time and to handle unsolicited keep-alive messages. The fifth section addresses the formats and protocols relevant to the EE caching function and the cache manager that is responsible for managing the persistent cache on both the EE client and the EE server. The sixth section reports on experiments to measure EE performance both in terms of data reduction and response time. Finally, the conclusions are stated.

Overview of Emulator Express data reduction

As shown in Figure 2, EE uses a dual-proxy configuration that consists of a *Client Side Intercept* (CSI) program located in the mobile unit and a *Server Side Intercept* (SSI) program located in the wireline network typically at or near the host system. This dual-proxy approach borrows from the IBM eNetwork WebExpress model[10–12] for optimizing Web browsing over wireless networks. The CSI-SSI pair is transparent to both the client's emulator and the Telnet server (and host application). The emulator communicates with the CSI via a TCP/IP connection using the TCP/IP loop-back feature that enables TCP/IP communication without the need for data to pass over an external communications adapter. The CSI and SSI communicate using a private protocol over a TCP/IP connection. The wireless link exists on this connection, and CSI and SSI cooperate to perform data reduction. The emulator data stream seen by the emulator and the Telnet server is the same as though

Figure 2 Dual-proxy model

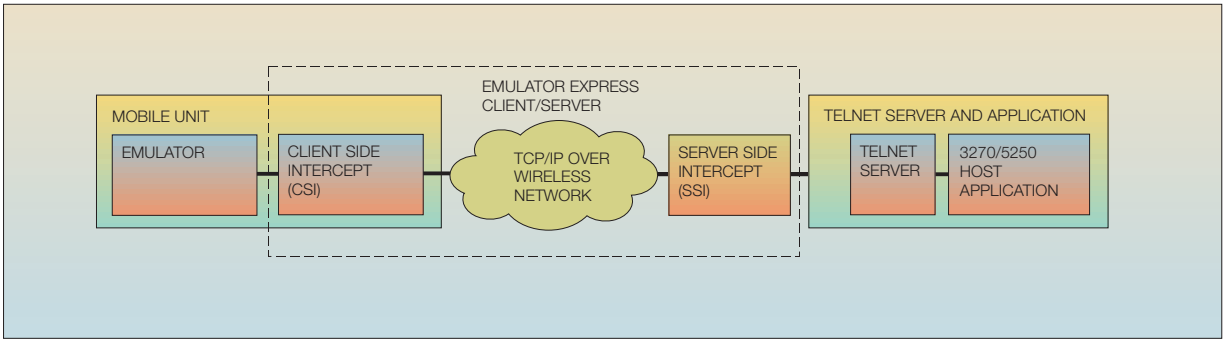
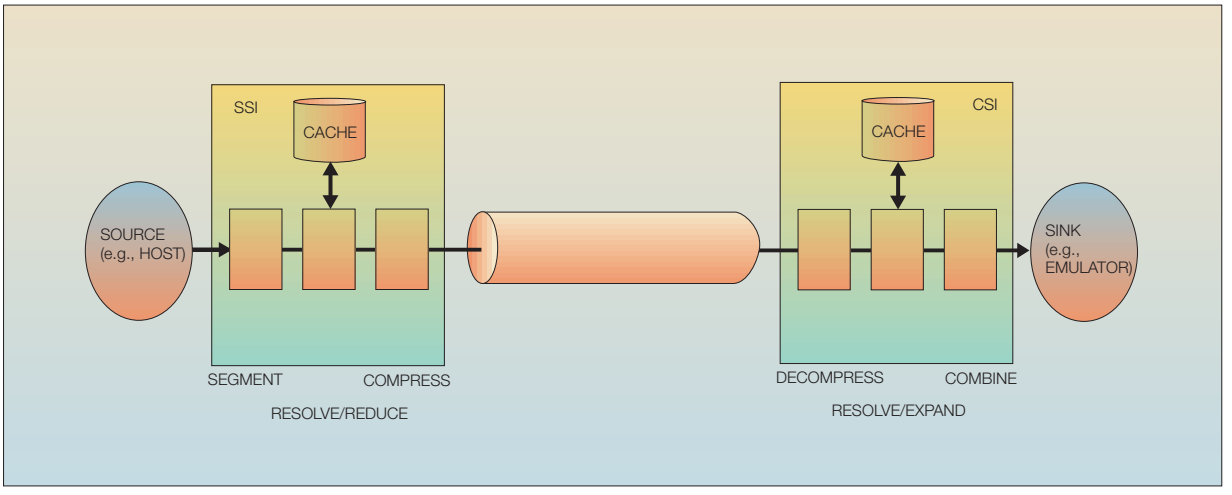


Figure 3 Basic data stream caching model



there were a direct Telnet connection between the emulator and the Telnet server; i.e., the CSI reconstructs data received from the SSI into valid Telnet data. Similarly, the data stream received by the SSI from the CSI is recomposed into a valid Telnet data stream that is delivered to the Telnet server. Therefore, except for minor configuration changes, the emulator and Telnet server are completely unaware of the presence of the EE system. EE configuration is discussed in more detail in the next section. Although not shown in Figure 2, many mobile clients may be connected to a single SSI; likewise, a single SSI may communicate with many different Telnet servers (and application hosts).

Data stream caching, a new technology. As shown in Figure 3, the heart of the EE system is a combination of caching and compression technology that significantly reduces the large outbound (host-to-client) data streams typical of 3270 and 5250 sessions. Most 3270 and 5250 applications use menus or other screen formatting techniques that result in parts of the displayed image such as prompts or instructions to recur many times in a typical application. The EE caching technology does not attempt to maintain a screen image, but rather uses knowledge of the appropriate 3270 or 5250 data stream protocol to break an output data stream into segments that are likely to have high probability of being reused on a later

screen, or possibly even the same screen. The segments are cached at both SSI and CSI so that when a segment recurs, only the segment number is retransmitted to the CSI. By combining this caching with a more traditional compression approach, EE is usually able to reduce outbound data streams by a factor of five or more. The data stream that is transmitted over the radio network is thus characterized by transmission of significantly less data and, usually, fewer packets. Periodically, a snapshot of the current caches is saved as a checkpoint. Later sessions may use the last successful checkpoint as a starting point for the caches to be used in that session. The caching technology is discussed in greater detail in the fourth section.

Basic flow. Let us consider a typical outbound data stream. There will usually be many fields on the screen containing static text that is used to prompt for input. Some input fields may be preloaded with data that can be changed by the user. The SSI will analyze this data stream and divide it into segments. A key field is generated for each segment. If the cache already contains the segment, then the two-byte block number replaces the segment data in the output data stream. Otherwise, the new data are both cached for possible future use and forwarded to the CSI. An LRU (least recently used) algorithm is used to discard the oldest segments when the cache becomes full.

The compression logic works on the data buffers to be sent between the SSI and CSI. Unlike caching, compression in EE operates in both directions and starts afresh each time a new session is established. Many compression algorithms are readily available,[13] and the choice depends on the overall system objectives (compression ratio versus computing cycles). We found arithmetic compression to be extremely effective for 3270 and 5250 emulator data streams containing, as they do, mostly textual data. Arithmetic compression may consider only the current character in the input stream (order-0 compression) or the previous n characters (order-n compression). Higher-order compression usually does a better job of compression but requires more memory and more CPU cycles than lower-order compression. Arithmetic compression was chosen for EE because, in a wireless environment, the better data reduction was deemed more important than saving CPU cycles or memory. The choice was made to use arithmetic order-3 compression for EE to provide aggressive compression within the typical CPU and memory constraints of contemporary mobile devices. A candidate extension for EE is to permit additional compression options, either different orders of arithmetic compression or different schemes altogether.

The output buffer is compressed, and the result is then sent over the CSI-SSI TCP/IP connection. Surprisingly, we found compression to be effective even when most of the buffer consists of segment addresses (in contrast with the actual segment data). This occurs because there is substantial redundancy in the segment control headers that are required to delineate segments in the data stream.

At the EE client, the incoming data stream is first decompressed. The caching instructions in the decompressed data stream are now analyzed to reconstruct the original data stream. Cached segments are retrieved, and new segments are stored in the cache as appropriate. In this way, the EE client maintains a mirror image of the server's view of the cache.

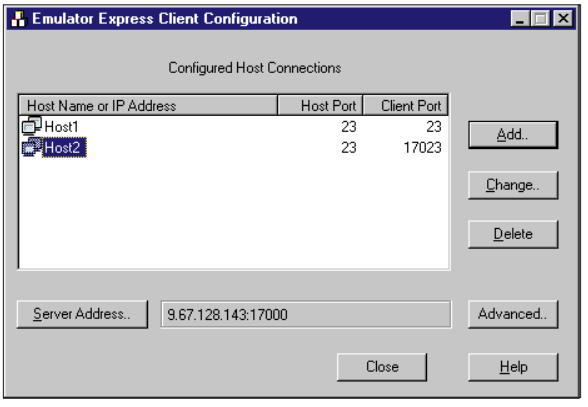
Emulator Express configuration

Because the EE system uses intercept programs, a certain amount of configuration is necessary. The Telnet emulator program is configured to communicate with the CSI rather than the ultimate Telnet server. This communication is entirely within the mobile device and is done in memory using the TCP/IP loop-back feature. Similarly, the CSI must be configured to connect to the appropriate SSI. The SSI uses a single port to listen for incoming connections from clients. During session establishment, an initial packet containing the destination Telnet address and port number is sent from the CSI to the SSI, so that the SSI can construct the final connection to the destination Telnet server.

Figure 4 shows the client definitions for two application hosts, Host1 and Host2. The host name (or address) and port number define the location of the destination Telnet server. The *client port* specifies the CSI listening port for all sessions initiated to a particular host. This port may or may not be the same as the actual port used by the host applications. For example, the host and client ports are the same for sessions to Host1. However, the client port for Host2 (17023) is different from the host port (23). This is necessary in order to distinguish the sessions among the respective hosts.

The client port number is also sent to the SSI in the initial packet. As we shall see in the fifth section, the client port number is used to anchor a pair of syn-

Figure 4 Client configuration



chronized cache instances associated with sessions to a particular host.

In summary, when the CSI receives a connection on a given listening port, it establishes a connection to the SSI and sends an initial packet containing the client port number and the host Telnet server address and port number. Upon receiving the initial packet, the SSI establishes a connection to the host Telnet server, and the communication channel from the emulator to its Telnet server is now complete. This *circuit* of the three TCP/IP connections is maintained for the duration of the Telnet session. The only configuration at the SSI (except for trace flags) is the listening port number for incoming CSI connections. All session configuration is done at the client system. This design minimizes the administration of SSI servers and gives end users the open-ended flexibility to configure their host connections as required.

Emulator Express and Telnet

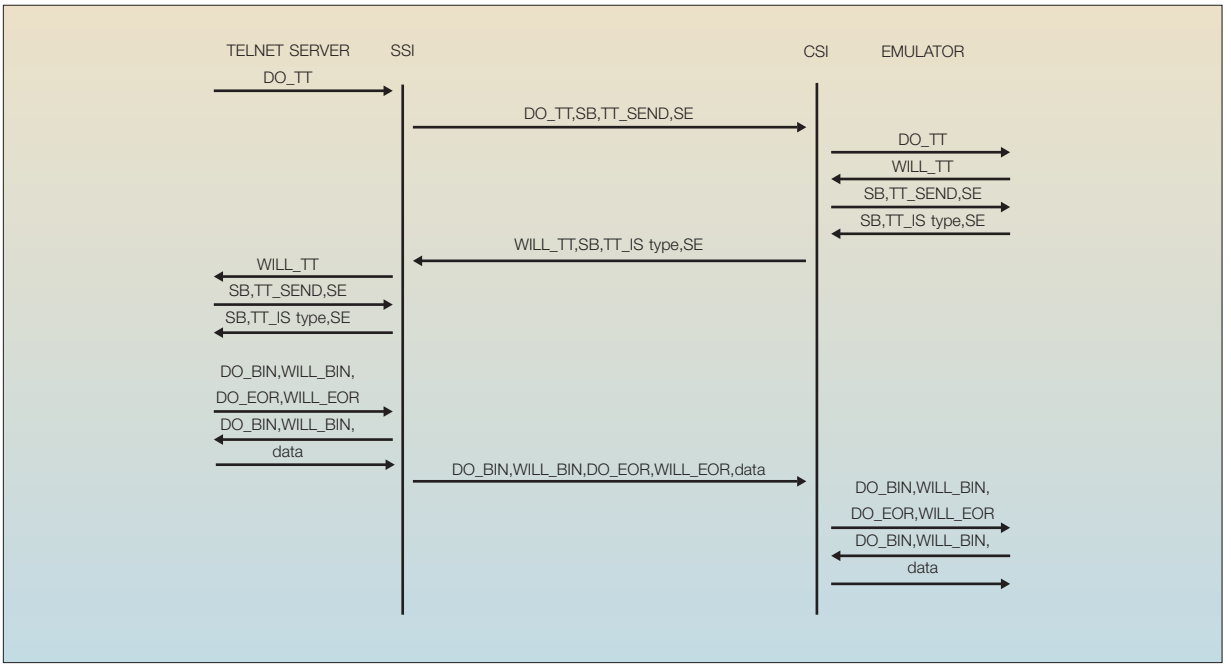
In this section, the TCP/IP and Telnet communications aspects of EE are discussed. Other optimizations implemented to improve initial connection time and to handle unsolicited keep-alive messages are also discussed.

TN3270 and TN5250. The Telnet protocol is defined in IETF Request for Comment (RFC) 854[14] and was originally designed to allow a Network Virtual Terminal (NVT) to access applications across the ARPAnet established by the Advanced Research Projects Agency. Because of the relatively high cost of 3270 or 5250 terminals compared to the limited-function ASCII terminals frequently used with minicomputers, several manufacturers introduced protocol conversion products that would allow an ASCII terminal to connect to the protocol converter which, in turn, was connected to a host network using BSC or SNA protocols. Increasing acceptance of TCP/IP networks led to the idea of combining the Telnet function with protocol conversion and produced Telnet 3270 (TN3270)[15–18] and later TN5250.[19] These combinations provide the display protocol conversion (emulation) at a Telnet client while providing a centralized communications protocol conversion to access the host BSC or SNA network at a Telnet server. The 3270 or 5250 data stream is thus transported intact across the IP network via TCP/IP. With the rise in popularity of personal computers, it became common to have the Telnet client actually run on a PC and present the data either to the character-based console or in a graphical window on the PC desktop. Similarly, as TCP/IP connectivity moved into the mainframe computer and AS/400* systems themselves, the Telnet (TN3270 or TN5250) server could also be implemented within these larger systems, removing the need for a separate processor and communication line.

Communication between a Telnet client and server is initiated by the client opening a TCP/IP socket to a port on the server. The assigned well-known port for Telnet is 23, although it is possible to use a different port, for example, if it is deemed desirable to have two different Telnet servers on a system. The user at the client end must have a method of specifying the IP address or name of the Telnet server machine and also the port to use. Additional configuration options are typically available to the user, including the ability to designate an SNA logical unit (LU) name to be used within the SNA name space of the host system or features of the terminal being emulated such as number of rows and columns.

Telnet sessions using circuits of TCP/IP connections. With the addition of the two intercepts in the system, the single Telnet connection, or session, is now replaced by three connections: a connection from the emulator to the CSI, a connection from the CSI to the SSI, and a connection from the SSI to the Telnet 3270 or Telnet 5250 server or host system. The connection between the two intercepts is implemented in the present design with one TCP socket connection for each Telnet session. This implementation allows each session to be managed by its own thread or process independently of the other sessions running on the same processor. Alternate ap-

Figure 5 Emulator Express Telnet reduction



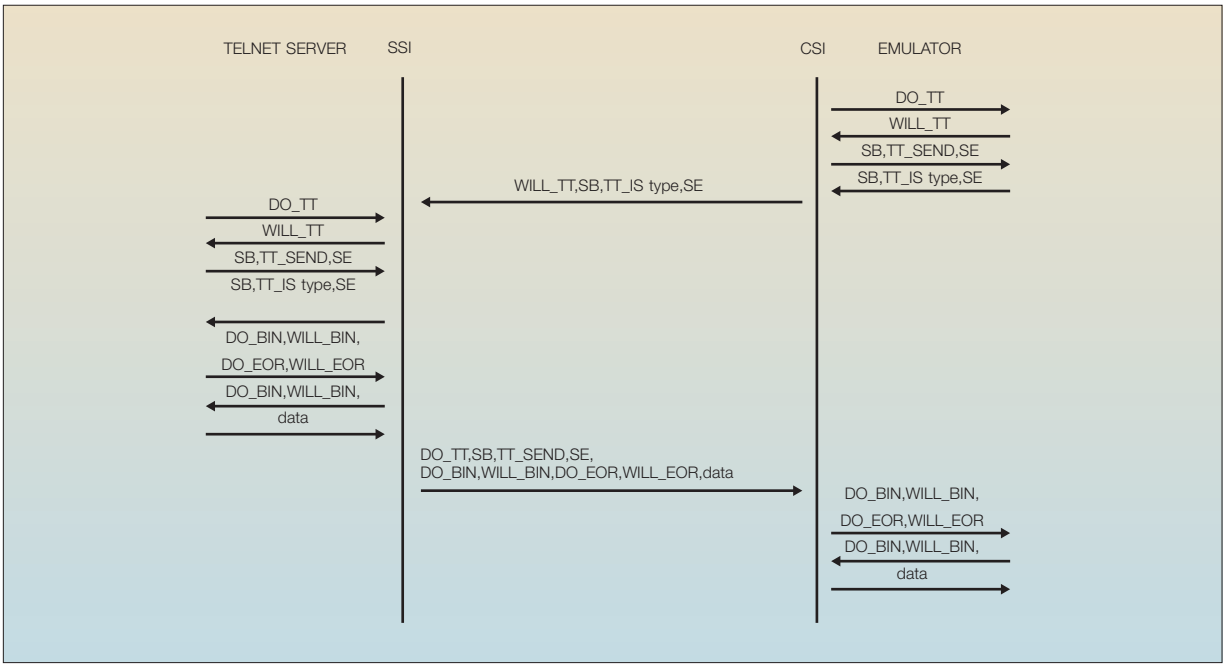
proaches might allow all sessions to be multiplexed over a single pipe between the two intercepts. Although TCP is used between the two intercepts in the present design, another protocol could be substituted if it were more suited to the underlying transport medium. Indeed, a protocol based on the User Datagram Protocol (UDP) was tried early in the development cycle. However, the small performance gain achieved was offset by loss of reliability and less stable performance as compared with TCP.

Telnet protocol reduction. In order to speed session startup, some optimization of the standard Telnet negotiation protocol is done between the client and server-side intercepts. This optimization is accomplished by presuming that if terminal negotiation is requested, it will indeed be performed, and by presuming that if 3270 or 5250 terminal types are negotiated, the corresponding binary and the end-of-record negotiations will also be successful. Figure 5 shows the reduction in number of flows using this approach. Figure 6 shows a further possible optimization where the client does not wait for the server to start the negotiation but instead forwards the information for the previous successful negotiation presuming that the same negotiation will occur again.

The reduction in number of flows is clearly shown by comparing the exchanges between CSI and SSI with those between CSI and emulator or SSI and Telnet server. Reducing the number of exchanges in this way makes the session startup significantly faster than it would be using the normal number of exchanges. This increase in startup speed can be particularly important for applications where the user will make a connection to interrogate the host system or report status and then disconnect after a relatively short session. A side benefit resulting from using networks that impose an additional charge for each packet sent is a reduction in the number of packets required for session startup and thus in cost.

Optimizing TCP "keep-alive" messages. An additional optimization is performed by the SSI that responds to keep-alive messages (such as timing marks) without forcing them to the client side. Since mobile clients may frequently move in and out of range, this optimization has the additional benefit of preventing the session termination of a temporarily unreachable client as well as the obvious one of reducing the air traffic. A drawback of this approach is that the SSI will not detect loss of connection with a client unless that loss is reported by the underlying

Figure 6 Further optimized EE Telnet negotiations



transport mechanism. This drawback is particularly detrimental where a client is using a particular LU. For example, if the client-side system is restarted while out of radio range and the server does not detect the session loss, then every connection attempt using the client LU name will fail until the original session is deactivated by some means or another. To address this problem, the CSI reports what sessions are still active whenever a new session is established. This information allows the SSI to terminate any sessions that are no longer active. The current system terminates all three sockets for each connection if any one terminates. It does not attempt to maintain the sockets between client and CSI or between SSI and server if the socket between CSI and SSI is terminated abnormally. The ability to survive such outages would be a possible enhancement.

Emulator Express caching

One challenge to developing data stream caching is determining what unit (object) is to be cached. Unlike other caching contexts (e.g., Web browsers, CPUs), emulator data streams do not consist of a series of uniquely identifiable objects, because the emulator and the host application maintain state information and cooperate to interpret the data. The objective was to determine meaningful segments that would be large enough to obtain a significant reduction in data if they are cached and also have a high access frequency. As noted earlier, the EE caching technology does not attempt to maintain a screen image, but rather uses knowledge of the appropriate 3270 or 5250 data stream protocol to break an output data stream into segments that are likely to have a high probability of being reused on a later screen, or possibly even the same screen. The segmentation algorithm is thus a *function of the data stream* being processed—in our case TN3270 and TN5250 data streams. In principle this *caching technology can be applied to many other data streams* by providing segmentation functions that are specific to the particular data stream.

The segments are cached at both SSI and CSI so that when a segment recurs only the segment number is retransmitted to the CSI. As illustrated in Figure 3, there are two persistent caches for each emulator session: one at the SSI and one at the CSI. Making the caches persistent allows EE to adapt to the usage pattern of an individual and provide improved performance on subsequent connections. Unlike tra-

ditional buffer compression techniques, the EE persistent cache avoids having to relearn the usage pattern each time a new session is established.

A 32-bit CRC (cyclic redundancy check) is generated for each segment and used as the key field. EE caches at most 32767 segments, so the use of a 32-bit CRC makes it extremely unlikely that two different segments can have the same key. If a segment with the same key as a new segment is already cached, the cached segment is compared with the new segment. If the segments match, the segment is replaced in the outbound data stream with an instruction to access the segment from the cache of the CSI. If there is no match for the key, the new segment is inserted in the cache. If the key is found but the input segment does not match the cache segment, the existing cache segment is deleted, and the input segment is inserted in the cache (with the same key). Whenever a new segment is inserted in the cache, the segment data are emitted to the output buffer with an instruction to insert the segment in the cache at the CSI. In order to avoid unnecessary wasted space, the EE server will only cache segments greater than a predefined minimum size. Segments that are smaller than this minimum are replaced in the outbound data stream buffer with an instruction to copy these data without caching. The process of resolving segments generates output data stream buffers to be sent to the client (CSI).

Session establishment and active cache creation. When a session is first established between a given mobile client and host application, a persistent cache is allocated at the CSI and SSI, respectively. It is called the *active* cache because it is accessed while the session is active. The first time a session is created the active caches are empty. Over the lifetime of the session, segments are stored and retrieved from the active cache; the cache becomes increasingly populated, and the degree of data reduction increases since there will be more "cache hits." For caching to work correctly, it is necessary that the active cache on the CSI and SSI remain perfectly synchronized. If the SSI replaces a data segment with a cache reference, it is necessary to guarantee the availability of the cached entry at the CSI so that the segment data can be retrieved. One possible alternative would be to devise a recovery protocol that would request the partner to send the data in the event of a cache failure. This alternative, however, requires many additional message flows, adds much complexity, and was deemed unacceptable in the wireless environment where the goal is minimizing data flow.

For subsequent sessions, however, we would like to use a cache that was generated during a previous session between the same client and host. To reuse caches across session instances requires that the active caches be saved with complete integrity before the session terminates. However, what if the session is abruptly terminated because of loss of signal, network outages, or a power loss? Abrupt disconnections are normal in a wireless environment. Following a sudden failure, the active cache must be considered useless because its state is unknown; segments in transit or in memory buffers will be lost. To guarantee the generation of consistent versions of the cache, a checkpointing protocol was developed.

Checkpointing. To cope with the prospect of unexpected failures, it is necessary to periodically generate a consistent copy of the active cache, called a *checkpoint*. The criterion for when to take a checkpoint is a function of session activity and time. For example, if the checkpoint time interval has passed but no activity has occurred on the session, no checkpoint is taken.

Taking a checkpoint. When the CSI determines that a checkpoint is necessary, it calls the cache manager to *prepare* the active checkpoint. The active cache files are locked, dirty pages are flushed to disk, the active cache files are copied to temporary files, the cache state is set to *prepared*, and finally the active cache files are unlocked so that session activity may continue. After the cache of the CSI is prepared, a checkpoint request is sent to the SSI; this command is "piggy-backed" on the next message sent to the host to avoid an extra message flow. The SSI first prepares its cache in the same manner as the CSI and immediately commits to create the next cache checkpoint. Commit simply consists of renaming the temporary files created during the prepare step and changing the cache state to reset. At any given point in time there may be two checkpoints labeled CP0 and CP1. Two checkpoints are necessary to ensure that there is always a valid checkpoint even if a failure occurs during checkpoint creation. After the checkpoint is successfully created, the SSI returns a positive acknowledgment to the CSI consisting of the checkpoint time stamp and a checkpoint number (0 or 1). As with the checkpoint request, the acknowledgment message is piggy-backed on the next message sent from the host to the client. When the CSI receives the positive acknowledgment, it creates its next checkpoint in like manner as the SSI except that it tags its checkpoint with the checkpoint time stamp and checkpoint number received on the checkpoint

response from the SSI. The CSI also sets state information indicating that a confirmation response to the SSI is pending. A confirmation message is sent from the CSI to the SSI to confirm successful generation of the checkpoint on the client device. Now

During session initiation it is necessary to determine which cache checkpoint to activate.

we are guaranteed to have identical consistent cache checkpoints at both the SSI and the CSI. When the SSI receives the confirmation message, it deletes its oldest checkpoint, thereby freeing up disk space.

Storing checkpoints. If we wish to use the same cache across multiple session instances, it is necessary to associate the checkpoint files with a specific client-host configuration. To accomplish this association, checkpoint files are anchored to the *client port* (described previously) at both the CSI and the SSI. For each client port, a state file and the checkpoint files are saved. Since a single SSI may service many clients, a client port alone is not sufficient at the SSI to associate checkpoint files with a specific session. In addition, a unique client identifier (ID) is required. Each checkpoint consists of a data file that contains the cached data segments and an index file keyed by the computed CRC value of a segment for fast access. At any point in time, one or two checkpoints may exist.

Client ID. At first glance, deriving a unique client identifier seems trivial because a client is uniquely identified by its IP address. However, this alternative fails to take into account that dynamic address assignment (DHCP) is typically used to assign IP addresses for mobile client devices, for example, using the PPP or SLIP protocols. If the client IP address is dynamically assigned, then the IP address is useless for uniquely identifying the client device across modem connections (wireless or wireline). To overcome this problem, a unique persistent client ID is generated by the SSI the first time a client device connects. Currently, the client ID is a four-byte entity composed of the three low-order bytes of clock time and a one-byte sequence number. For multiple server (SSI) support, a 16-byte globally unique identifier (GUID) could be used. When a session is initiated, the CSI sends a *SelectCheckpoint* command. Initially, it contains a null client ID. When the SSI receives the null client ID, it generates a unique identifier and returns it to the CSI. Client IDs are made persistent at both ends. If a client ID received by the SSI does not exist, the SSI reuses the ID; the server does not allocate a new ID. At the SSI, the client ID and the client port number are used to uniquely determine the cache checkpoints for a given client-host connection.

A practical problem with client IDs occurs when an installation replicates an existing hard drive to create a new system. This problem was not considered during the design of EE but occurs frequently in practice. If the generated client ID is not first removed from the hard drive, multiple actual clients may share a common, supposedly unique, identifier. The server cannot distinguish between a new session from a cloned ID and an undetected restart of the original owner of the ID, and the results are unpredictable. The automatic assignment of a new client ID to one of the offending systems is a possible enhancement.

Checkpoint selection during session establishment. During session initiation it is necessary to determine which cache checkpoint to activate. When a mobile user starts a session with a particular host application, the CSI must identify the checkpoint that was created during a previous session with the same application (i.e., via the same client port) and then communicate this checkpoint identifier to the SSI so that it can activate the corresponding checkpoint instance.

Activating a checkpoint. During session startup, the CSI activates its most recent checkpoint with the expectation that the SSI can do likewise. Activating a checkpoint means that the checkpoint files are copied to temporary files that will serve as the active cache during the lifetime of the session. The checkpoint files themselves are never modified once created. Next, the CSI sends a SelectCheckpoint command to the SSI and sets its state to *checkpoint response pending*. Upon receipt of a SelectCheckpoint command, the SSI attempts to locate the checkpoint identified by the time stamp and checkpoint number in the command. Theoretically, by design of the checkpointing protocol, the SSI should always find the checkpoint specified in the SelectCheckpoint command. If found, a positive response is sent to the CSI. If, for some reason (e.g., media failure) the checkpoint is not found, the SSI returns a "Not-

Found" response to the CSI, and creates a fresh active cache. When the CSI receives the NotFound response, it also starts with a fresh active cache, and processing continues. Eventually, checkpoints will occur to generate new checkpoint instances.

Multiple sessions to the same host. On VM (virtual machine) and MVS (Multiple Virtual Storage) systems, some users may have more than one user ID and, therefore, can have multiple concurrent sessions with the same host. With AS/400 systems, users may have multiple sessions for the same user ID. With modern emulators these sessions appear in separate client windows. Therefore, it may be quite natural and productive for a user to interactively switch among various host applications by clicking different emulator windows on the desktop.

One means for implementing multiple sessions to the same host would be to specify multiple-session configurations to the same host and assign different client port numbers for each. However, this alternative was rejected because it substantially increases client configuration and resource usage. It would be necessary to define multiple emulator sessions and their corresponding CSI definitions (as shown in Figure 4). Each defined session would require its own set of cache resources and, thereby, could substantially increase the amount of disk space required. We developed a much more user-friendly and resource-efficient solution that allows the creation of multiple concurrent sessions to the same host using a *single* client port number. However, this approach raised a number of challenging questions: Does each such session have its own cache and checkpoints? How are these sessions identified and bound to the proper cache? Must the checkpoint activity be synchronized across the different sessions? Are any additional configuration parameters needed?

All sessions between the same client and host that have the same client CSI listening port share common cache checkpoint files. However, each session gets its own active cache when the session is started. Checkpointing on each session is done independently. The last session to execute a successful checkpoint creates the latest checkpoint. However, since the checkpoint files are shared, race conditions can occur either when one session is being established and another is checkpointing or when more than one session is attempting to checkpoint at the same time.

One of the goals of our design was to never require one session to wait for the completion of a checkpoint in progress on another session. In a wireless environment this wait could take seconds. The only time a checkpoint is not available for activation is when it is being created. The CSI and SSI allow for two checkpoints to exist: CP0 and CP1. Persistent state information is maintained that records the checkpoint states for a given client ID. A labeling scheme is used to guarantee that at least one checkpoint is available for activation. Only one checkpoint is permitted to be in progress at any given time, ensuring that at least one of the two checkpoints will be free (unlocked) at all times.

If CP0 is the oldest checkpoint, then the next checkpoint request will result in the construction of a new CP0 (CP0′), and CP1 will be marked as the oldest checkpoint. CP1 will be activated as the active cache for any new sessions that are started during the checkpoint processing. The files associated with CP0 are deleted when the prepared cache is committed. At the CSI, after the positive acknowledgment is received from the SSI and the new checkpoint is committed, the files associated with CP1 are deleted. At the SSI, the files associated with CP1 are deleted when a confirmation message is received from the CSI as described previously. The CP1 files are retained until confirmation is received in order to guarantee that there is a valid checkpoint pair at the CSI and SSI even in the event of checkpoint failure.

If a CSI attempts a checkpoint and checkpoint processing is in progress for another session with the same client port number or the checkpoint is being activated for session startup, the checkpoint request is simply aborted, and normal processing continues. This polite, laissez-faire philosophy toward checkpointing is based on the recognition that the exact timing or even the content of a checkpoint is not crucial as long as the checkpoints are created with reasonable frequency. What is crucial is that the checkpoint protocol guarantee that the checkpoints created at the CSI and SSI are perfectly synchronized and that the protocol does not introduce noticeable delays in normal session traffic. In all likelihood different sessions will win the checkpoint race over a reasonable period of time.

Implications. The above design has the virtues of simplicity, nonblocking, and continuous availability. It is guaranteed that session initiation is always possible, that very few resources are locked for very long, and that no waiting is required for cache checkpointing. In the event of failure to activate a checkpoint, processing starts from scratch with empty caches.

However, at first glance it seems to have one major deficiency. Namely, if we assume that different sessions are dedicated to different host applications, one might expect that the cache buildup for one application would not be effective for use by other concurrently running applications. In the above design, the latest checkpoint would reflect the activity of the application running on a given session. This has not proven to be a problem in practice, possibly because there are many common segments across disparate applications (e.g., log-on screen). If it is important for an application to have its own dedicated cache, the user can simply configure a separate session to the same host with a different client port number. Alternatively, assuming a common client listening port for all applications to the same host, a user may initially execute his or her suite of common applications sequentially, and the effect will be to form a union of the respective caches. This might be best done in a wireline modem or LAN environment to *prime* the cache. Subsequently, when multiple sessions to the same hosts are constructed in a wireless environment, the cache will contain a representative set of segments for all the applications. Over time the cache will become biased toward the most frequently used application—a desirable effect.

The cache manager. The cache manager is the software component that allocates and deallocates cache space and stores and retrieves variable-length segments to and from the cache. This cache had to meet a number of requirements:

- Ability to store variable-length segments (from 16 to 4000 bytes).
- Efficient random keyed access to the data. Each segment is typically small (e.g., 100 bytes). It is important to quickly access a given segment as a function of its key and minimize disk I/O, particularly for frequently referenced segments.
- Bounded in size and self-organizing. Since a session may run an indeterminate length of time, a means to reclaim space had to be provided when the cache becomes full. An LRU algorithm was implemented to delete the oldest segments when the cache became full.
- Persistent across sessions. The cache data had to be persistent so that the benefits accrued from caching on one session (from a given client/SSI/Telnet server) could be carried over to the next session.

The main components (objects) of the cache manager are shown in Figure 7.

An EE cache consists of three files: a *segment store*, an *index store*, and a *state file*. Each store is comprised of a set of fixed-length blocks, and *BlockManager* is the base class that is responsible for managing the blocks. The state file pertains to cache checkpointing discussed above.

The *SegmentCache*, a subclass of BlockManager, supports the ability to store variable-length data segments using one or more blocks of the segment store. In object-oriented terminology, a segment store is an *instance* of the SegmentCache class.

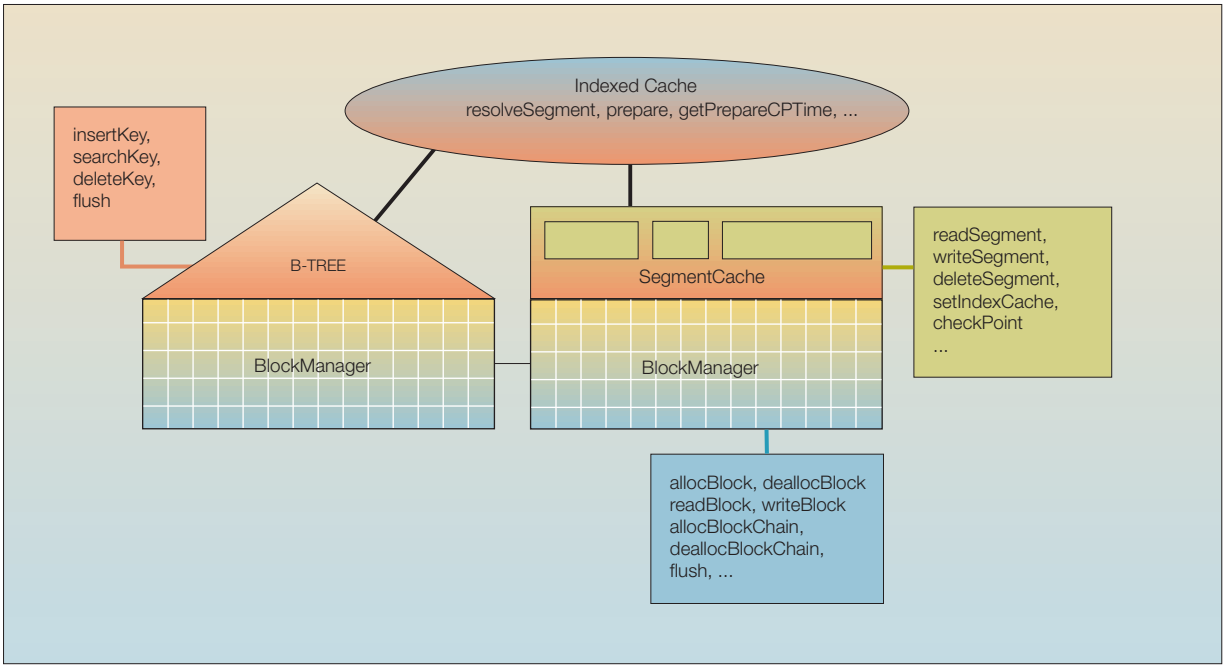
The B-Tree class, also a subclass of the BlockManager, implements a B-Tree index, where each block corresponds to a B-Tree node. Each B-Tree entry is composed of a segment key and the address of the segment in the segment store. The segment key is a 32-bit CRC value computed for its respective segment. The address of a segment is the block number of the first block of a segment in the segment store.

The *IndexCache* is a container class that consists of a B-Tree object and a SegmentCache object. The IndexCache provides the interfaces to the CSI and SSI components that access the cache. It is responsible for coordinating the activity across the segment and index stores; e.g., the index must be updated when a new segment is written.

BlockManager. The primary methods of the BlockManager are shown in Figure 7. When a block is allocated, it is marked *in use*, and ownership is transferred to the calling application. When a block is deallocated, ownership is transferred back to the BlockManager (i.e., placed on a free-block list). A BlockManager store is pageable. Blocks are paged to or from real memory so that file I/O is eliminated for blocks that are resident. Typically, around 20 percent of the blocks accessed require a file I/O operation. The *flush* method forces all "dirty" pages to disk whenever a cache checkpoint is required.

The block size and the number of blocks are input parameters to the BlockManager constructor. The segment store typically consists of many small segments. Each segment is stored in an integral number of blocks. And the block size is small (48 bytes) to minimize access time and reduce wasted space. There may be unused space in the last block of a variable-length segment. The block size for the index store is tuned to the size of the B-Tree node which is defined by the maximum number of keys (degree) per node. Currently the degree of the B-

Figure 7 Cache manager object



Tree is 15, meaning that a node can hold a maximum of 29 keys and have up to 30 successor nodes. Thus, the BlockManager permits the block access (and paging) to be tuned to the particular usage desired.

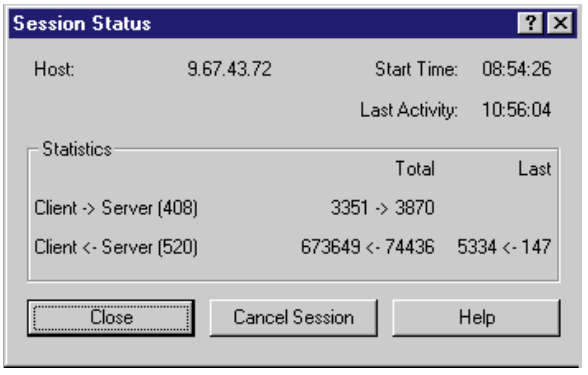
We wanted to avoid having to preallocate the maximum cache size permitted for all users. Some users may have more repetitive usage than others, and their caching demands may vary greatly. To handle this situation, we implemented a method to "grow" the cache incrementally. An initial allocation is specified when the segment store is created; subsequently, as the current allocation becomes full, space is incrementally added. This process repeats until a maximum allocation is reached, and then the LRU algorithm is used to free old segments when new segments are to be stored.

SegmentCache. In the SegmentCache, variable-length segments are allocated over one or more blocks. These blocks are chained together to form a *block-chain*. Each allocated segment is placed on an LRU chain when it is created (written). Each time a segment is referenced, it is moved to the head of the LRU chain. When the cache is filled, a predefined percentage of the oldest segments are freed; i.e., all the blocks in the block chain of each freed segment are deallocated. The *setIndexCache* method is used to bind a B-Tree index to the segment store. It is called when an instance of IndexedCache is created. The segmentCache object needs to be aware of its indexes so that the respective index entries can be deleted when a segment on the LRU chain is deallocated. It is possible to associate multiple indexes with a segment store, but we only use one.

IndexedCache. The IndexedCache class ties together the segment store and its index store. The creation of an IndexCache object causes an *active cache* to be instantiated either by copying the latest checkpoint files to working temporary files or by opening new files if no checkpoints exist. When an IndexedCache object is destroyed, the corresponding active cache files are deleted. That is, the currently "active" cache space is deleted. Persistence is achieved by periodically taking checkpoints, which guarantees saving a consistent copy.

The *resolveSegment* method accepts a data segment buffer and returns the address of the segment in the cache or a NOT-FOUND return code if the segment

Figure 8 Session status display



cannot be found. The resolveSegment processing includes: computing the segment key (i.e., CRC), searching the B-Tree, and writing the segment in the cache if it was not found. Writing the segment in the cache includes writing the data into the segment store and inserting the key and address of the segment in the index.

The *prepare* method is called to prepare the cache for making a checkpoint. The BlockManager flush method is called to force all dirty pages to the segment store and the index files, and the file flush operation is called to force all file buffers to disk. Finally, the state file of the cache is updated to indicate that the current active cache is in the prepared state.

Emulator Express performance

New users are often amazed at the responsiveness of applications when using Emulator Express. Comments such as "this is faster than my office LAN" are not uncommon, even though the user is using an application from a car-mounted mobile terminal.

EE includes a status display so that the user may display the data traffic counts for a session as shown in Figure 8. Both the raw counts sent to and from the application as well as the amounts sent between the EE CSI and SSI are shown. Total counts in each direction as well as the count and direction for the last message are displayed. The display shown in Figure 8 indicates total traffic from the host of some 674 kilobytes (KB), whereas only 74 KB were actually sent between EE SSI and CSI, or a reduction of approximately 9:1. Note that the last message of 147 bytes was all that was transferred instead of the 5334 bytes needed to display the screen shown in Figure 9.

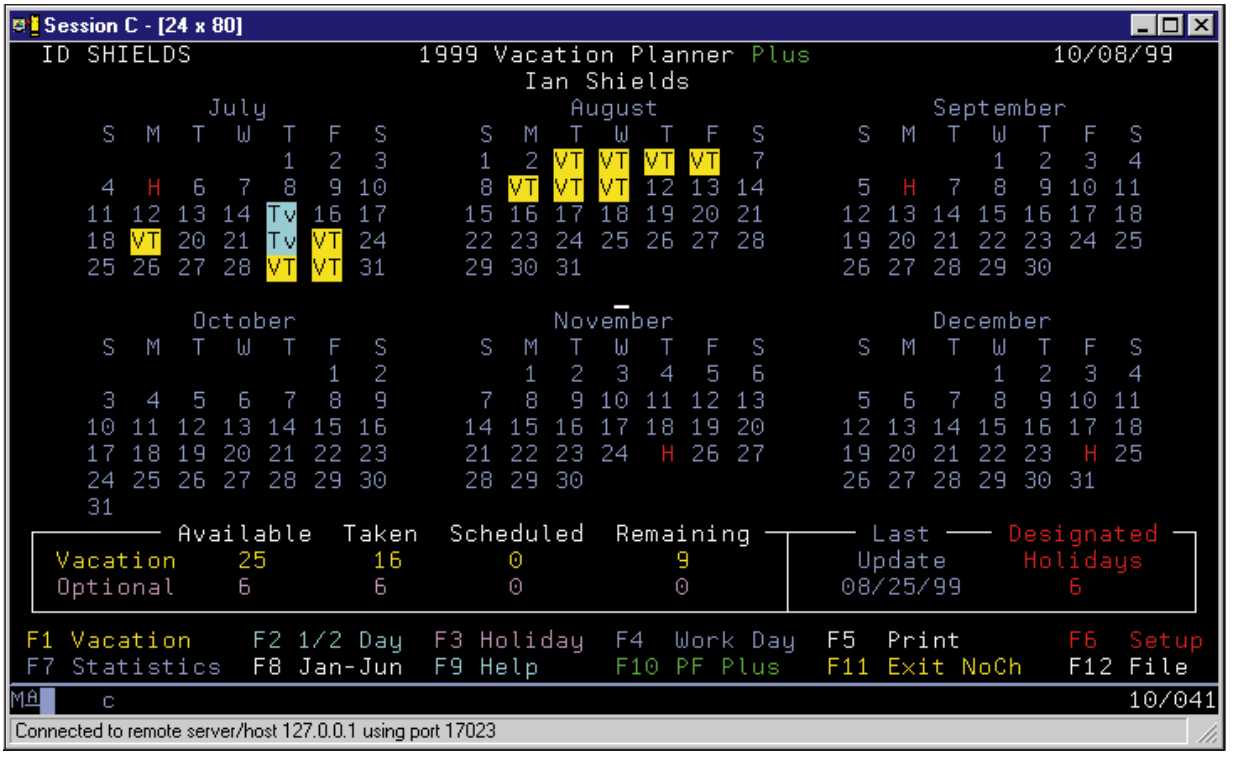
The reductions seen for different application sets vary widely. Ratios between 5:1 and 10:1 are common over a complete session, with occasional figures as high as the 36:1 measured above for individual screens with high amounts of repetition. Note also that inbound data from CSI to SSI may slightly increase where input is mostly an attention key with no user data input as is the case with the illustration shown here. However, the resulting data are still much smaller than outbound data, and any significant user input will generally see compression have a noticeable effect on inbound data.

The AS/400 system supports both 3270 and 5250 applications. We have done some limited measurements with scripts running the same application from both TN3270 and TN5250 connections and found that even though the 5250 connection results in as much as 50 percent more outbound data, EE reduces the SSI-to-CSI data transfer to about the same final amount regardless of whether the 3270 or 5250 data stream is being used.

To illustrate the effectiveness of EE, we performed some measurements with a script that does several telephone directory lookups followed by completion of a travel expense account. The script was recorded with the IBM eNetwork Personal Communications (PCOM) emulator program and includes delays between output and the following input such as occur in normal interactive usage. Response time summaries are shown in Figure 10.

We used the ARDIS packet radio (wireless) communications network and the IBM eNetwork Wireless Client Version 4 along with the IBM eNetwork Wireless Gateway Version 4 (predecessor products to the currently available SecureWay Wireless Client and SecureWay Wireless Gateway) for the radio runs, whereas we used the former IBM Global Network (IGN) at several different connect speeds for the dial-up runs. Runs were made with and without EE, and these are designated as *express* or *native*, respectively. Normally, the IBM eNetwork Wireless Client compresses transmitted data. For comparison we also did the native radio runs with this compression disabled. For all the runs with EE we disabled the compression in the eNetwork Wireless Client since EE already compresses the data. For the dial-up connections we did not do anything to disable any modem compression. All runs were made on a week-

Figure 9 3270 screen



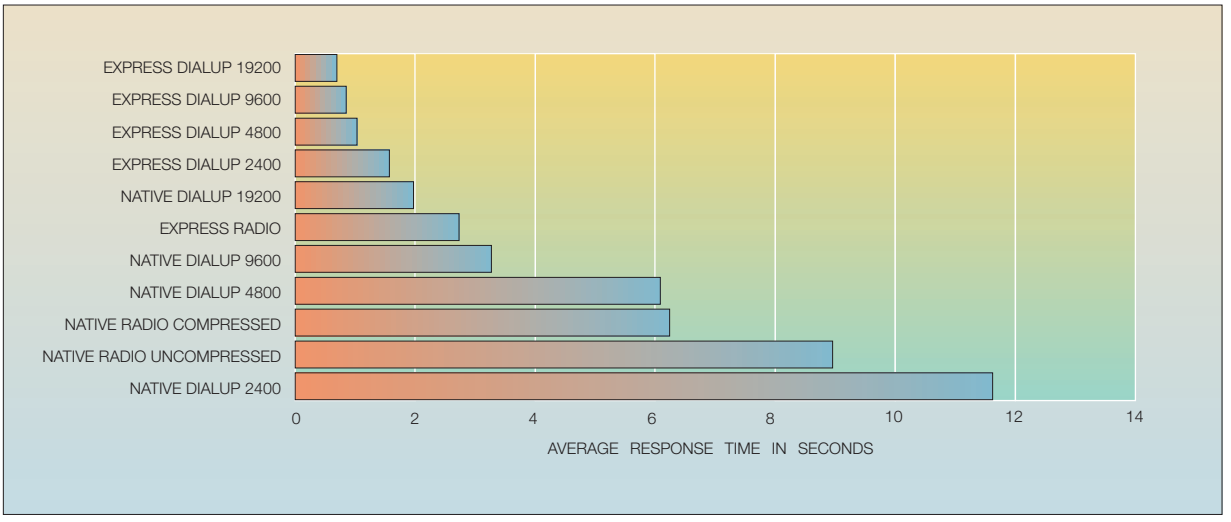
end or during off-peak hours to minimize the effects of heavy network congestion. Data shown are the average of two runs performed at separate times over a period of four days. The averages were also tabulated for each run, and the differences between each run of a pair were found to be quite small. EE runs were all made with an existing (warm) cache. Response times were obtained by analyzing trace data from the PCOM emulator and represent the response from an input action to the last output action for that input.

We have not attempted to quantify the impact of errors or congestion in a radio network on the EE user because such measurements would require additional information from the appropriate carrier. In normal use we have noticed degradation in response time on some networks during certain periods of the day. On some occasions this degradation was sufficient to prevent us from completing a script in native mode at all, although we were able to complete the script when using EE.

By reducing the amount of output data as much as it does, EE obviously reduces the time required to transmit messages and thus significantly improves response time as is evident in Figure 10. The script had approximately 60 interactions, so each second of response time saved is a minute of time saved for the total job.

Over slow links with variable delays, such as frequently seen with radio networks, this reduction in transmission time is doubly important. Delaying a short message for two or three times the transmission time does not usually trigger retry activity, whereas delaying a long message for even part of its transmission time may trigger retry activity that can easily spiral out of control. Thus, an EE user will see more consistent behavior and will also suffer less from extensive or irrecoverable retry activity. Figure 11 shows the distribution of response times using EE over the ARDIS network without eNetwork Wireless compression and also using a native PCOM session both with and without the eNetwork Wireless compression. A few responses for the native uncompressed connections fell outside the range shown here, being longer than 20 seconds, whereas approx-

Figure 10 Response time comparison



imately 75 percent of the EE responses were received within three seconds.

For the script that we used, approximately 135 KB of data are sent from the host for each session. The combined effects of EE caching and compression reduced this amount to just under 10 KB. The reduction of 13:1 was achieved here using the same data for each run. Actual production usage with 3270 applications has shown data reductions typically between 5:1 and 10:1. The cache files occupied approximately 220 KB of disk space and did not reach their configured maximum size of 512 KB.

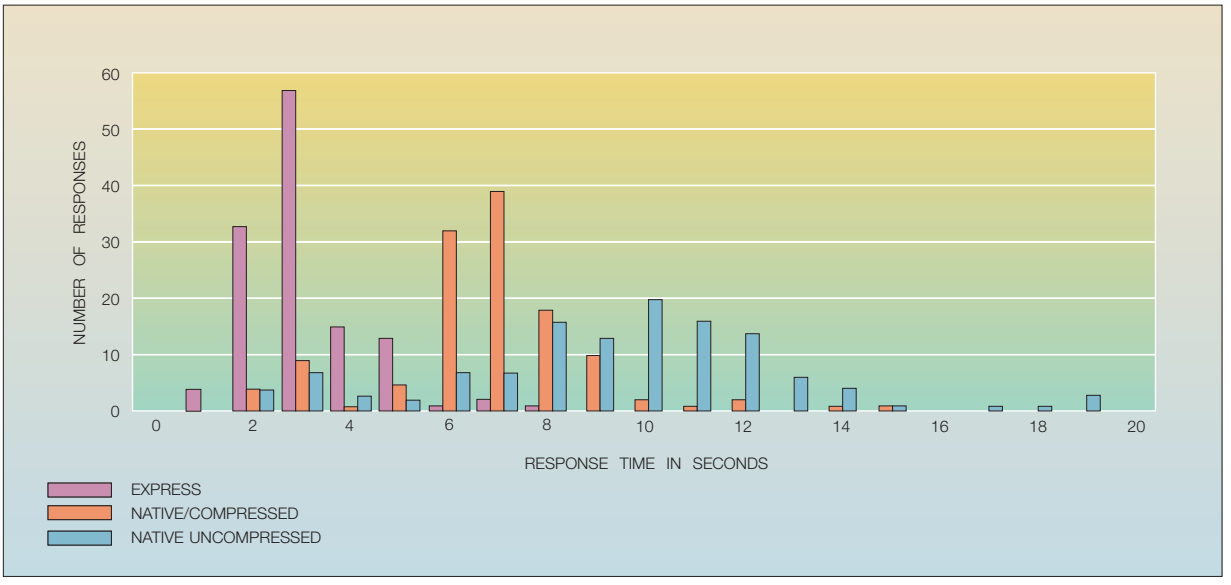
The results shown in this section are measurements of one environment. Many factors influence the performance of EE systems, including the speed and reliability of the connection between the client and the server systems. Other factors include the nature of the workload or application, the SSI load, the network load between the SSI and the host or Telnet server, and the host load, to name a few. This limited set of experiments represents only a small sample of the possibilities of EE. Actual measurements in other environments may vary.

Conclusions

This paper has described novel technologies that make it possible to run general 3270 and 5250 emulation from a mobile unit over very low-bandwidth wide-area wireless networks. These technologies (data stream caching and Telnet protocol reduction) combined with traditional compression are described in the context of their implementation in the IBM Emulator Express, part of the IBM SecureWay wireless product suite. Performance measurements show that Emulator Express enjoys a significant compression ratio for Telnet traffic and improves response times on links up to 56 Kbps.

This work raises the question: Should we investigate the possibility of adding new (optional) protocol to the TN3270 and TN5250 Telnet regimes that would support the Emulator Express optimizations? Such extensions would require emulators and Telnet servers to support the compression, caching, and possibly the protocol reduction functions provided in the CSI and SSI, respectively. There are arguments for integrating these functions into Telnet: First, these optimizations may become much more pervasive across the Telnet domain, since they would probably be implemented by various emulator and Telnet server vendors. Second, including these optimizations as part of Telnet would yield a more efficient system since the three TCP/IP connections required by EE (for a single session) would be replaced by a single connection. Third, these optimizations would work in concert with the transport layer security (TLS) enhancement recently added to Telnet.[20] TLS defeats the optimizations described here because the CSI and SSI require the emulator data streams to be in the

Figure 11 Distribution of response times over radio communications



clear. If the EE optimizations were built into the TN3270 and TN5250 protocols, the optimizations could be applied before and after the data stream is encrypted and decrypted, respectively.

A key advantage of the current approach is that a single implementation can offer optimization transparently to any emulator session; i.e., no changes to any emulator or Telnet server are required. We argue that adequate security in most practical cases is possible with the EE model if we encrypt the CSI-SSI session. Since the CSI is coresident with the emulator, there is negligible exposure of compromise over the emulator or CSI session. If we presume that the SSI resides in a secure domain, either colocated with the Telnet server or resident on the secure side of a firewall, there is minimal exposure of compromise for data flowing on the SSI and Telnet-server connection. As an aside, we note that the data stream on the CSI-SSI session consists largely of arithmetically compressed references to cached data and is not easily readable by a casual eavesdropper. Although this situation would not deter a serious attack, it may well be sufficient protection for some applications where absolute privacy is less important.

Perhaps the most far-reaching aspect of this work is the development of the data stream caching technology. In principle, this technology can be applied to venues other than emulation and, therefore, can contribute toward improving efficiencies for a wide range of distributed network applications.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc.

Cited references

1. *IBM SecureWay Wireless Software*, product documentation on the IBM SecureWay wireless software products, including SecureWay Wireless Client and Gateway and eNetwork Emulator Express, can be found at http://www.software.ibm.com/enetwork/mobile/library.
2. IBM Host-On-Demand Certified "100% Pure Java™" Web-to-host access software runs smoothly on all Java-enabled systems; see http://www.networking.ibm.com/netmsg22.html.
3. *CICS Transaction Gateway, Version 3.0, Administration and Programming*, SC34-5448-00, IBM Corporation (September 1998); see http://www.software.ibm.com/ts/cics/library/manuals.
4. *SecureWay Host Publisher, Version 2*, G325-3937-00, IBM Corporation (September 1999).
5. *IBM eNetwork Personal Communications Version 4.3 for Windows 95, Windows 98, and Windows NT Host Access Class Library*, SC31-8685, IBM Corporation (January 1999).
6. T. Brawn and S. Gunn, *Internet Draft draft-ietf-tn3270e-ohio-01.txt*, IETF TN3270E Working Group (April 1999).
7. T. Imielinski and B. R. Badrinath, "Mobile Wireless Computing: Challenges in Data Management," *Communications of the ACM* **37**, No. 10, 18–28 (October 1994).

8. *ARTour Emulator Express Server for AIX, Version 2*, GC31-8299-00, IBM Corporation (March 1996).
9. *ARTour Emulator Express Server for OS/2, Version 2*, GC31-8298-00, IBM Corporation (March 1996).
10. B. Housel and D. Lindquist, "WebExpress: A System for Optimizing Web Browsing in a Wireless Environment," *Proceedings of the Second Annual Conference on Mobile Computing and Networking* (1996), pp. 108–116.
11. B. C. Housel, G. Samaras, and D. B. Lindquist, "WebExpress: A Client/Intercept Based System for Optimizing Web Browsing in a Wireless Environment," *Mobile Networks and Applications* **3**, No. 4 (January 1999).
12. H. Chang et al., "Web Browsing in a Wireless Environment: Disconnected and Asynchronous Operations in ARTour WebExpress," *Proceedings of the Third Annual ACM/IEEE International Conference on Mobile Computing and Networking* (1997), pp. 260–269.
13. M. Nelson, *The Data Compression Book*, M&T Publishing Inc., New York (1992).
14. J. Postel and J. Reynolds, *Telnet Protocol Specification*, RFC 854, IETF Network Working Group, NIC 18639 (May 1983).
15. P. Rehkter, *Telnet 3270 Regime Option*, RFC 1041, IETF Network Working Group (January 1998).
16. C. Graves, T. Butts, and M. Angel, *TN3270 Extensions for LUname and Printer Selection*, RFC 1646, IETF Network Working Group (July 1994).
17. J. Penner, *TN3270 Current Practices*, RFC 1576, IETF Network Working Group (January 1998).
18. B. Kelly, *TN3270 Enhancements*, RFC 2355, IETF Network Working Group (January 1988).
19. P. Chmielewski, 5250 Telnet Interface, RFC 1205, IETF Network Working Group (February 1991).
20. M. Boe, *Internet Draft draft-ietf-tn3270e-telnet-tls-02.txt*, IETF TN3270E Working Group (July 1999).

Accepted for publication December 3, 1999.

Barron C. Housel *Chapel Hill, North Carolina (electronic mail: bchousel@yahoo.com).* Dr. Housel recently retired from IBM as a Senior Technical Staff Member. He joined IBM in 1964 after receiving an M.S. in engineering science from the University of Oklahoma. He received an M.S. in computer science from Stanford University in 1968 and a Ph.D. in computer science from Purdue University in 1973. He was active in the development of database technology with IBM Research in San Jose, California, from 1973 to 1977. During the years 1977 to 1978 he was a guest faculty member in the Computer Science Department at Purdue University. In 1979 Dr. Housel joined IBM in Raleigh, North Carolina, where he contributed to the design and development of SNA and networking products. He has been involved in the development of wireless products and technology since 1995. He was a member of the IBM Academy of Technology and is a member of the IEEE and ACM professional societies. Dr. Housel has 18 patent filings and 25 patent publications.

Ian Shields *IBM Pervasive Computing Division, P.O. Box 12195, Research Triangle Park, North Carolina 27709 (electronic mail: ishields@us.ibm.com).* Mr. Shields joined IBM in Canberra, Australia, as a systems engineer in 1973 where he worked on communications systems for several Commonwealth Government accounts. He moved to Montreal, Canada, in 1979 where he worked in the Communications Systems Marketing Center and the Eastern Region Field Support Center and developed an NCP user line control for Lotto Quebec as well as providing other communications support. He moved to Raleigh, North Carolina, in 1984 and rejoined IBM in 1987 at the Research Triangle Park laboratory where he is currently a senior programmer in the Pervasive Computing Division. Mr. Shields has worked on development of custom code as well as mainline products and has spent several years working on products designed for use with radio networks. He has several patent filings and three issued patents. He studied pure mathematics and philosophy at the Australian National University and graduated with a B.A. (Hons) in 1974. He received an M.S. in computer science from North Carolina State University in 1995 and is currently pursuing a Ph.D. there.