Multi Service Link Layers: An Introduction

George Xylomenos

October 2007

Abstract

This document provides an introduction to the Multi Service Link Layer framework, describing its components and their interaction. Besides generic framework components, such as the scheduler, multiplexer and demultiplexer, a description is given of the specific link layer protocols available. In addition, a collection of execution and result processing scripts are described. The document applies to all versions of the framework, indicating changes from version to version. An installation guide and a change log are included at the end of the document.

1 Basic Concepts

In the *Multi Service Link Layer* (MSLL) scheme, multiple link layer protocols are multiplexed over a single link, without being aware of each other. The goal is to use the right protocol for each application, for example, a partial recovery scheme for real time applications and a full recovery scheme for non real time ones. Each protocol must work in isolation, so as to employ protocol code similar to that of a simple link. Since we only have one physical link, we need code to:

1. Receive each packet from the IP layer, select the appropriate protocol, and pass the packet to it. The selection is made by using the DiffServ field, although other options are possible. This is the *classifier*.

2. Receive each ready packet from a protocol and queue it. Whenever the physical link is ready, select the next packet for transmission and transmit it. The selection requires an algorithm that will ensure that no protocol will abuse the link. This is the *scheduler*.

3. Receive each packet from the physical link, select the appropriate protocol, and pass the packet to it. In order to achieve this, the scheduler tags each packet with a protocol ID before transmission. This is the *demultiplexer*.

4. Receive each packet released from a protocol, and pass it to the IP layer. This hides the multiple protocols from IP. This is the *multiplexer*.

We can add to this infrastructure a number of link layer protocols protocols, programming the classifier to associate each DiffServ value with a protocol. Therefore, the life cycle of a packet from the time it is passed from the IP layer to the link layer on the transmitter side until the time it is passed from the link layer to the IP layer on the receiver side is as follows:

1. The IP layer passes the packet to the classifier.
2. The classifier looks at the DiffServ field of the packet and passes it to the appropriate protocol.
3. The protocol processes the incoming packet as needed (adds headers, keeps a copy, starts timers, and so on).
4. The protocol passes the packet to the scheduler for transmission.
5. The scheduler tags the packet with a protocol ID and puts it in a queue.
6. Whenever the physical link is available, the scheduler chooses the next packet to send and transmits it.
7. The packet is received on the other side and is passed to the demultiplexer.
8. The demultiplexer removes the protocol ID of the packet and passes it to the appropriate protocol.
9. The protocol processes the received packet as needed (removes headers, stops timers, generates ACK/NACK packets, and so on).
10. The protocol passes the packet to the multiplexer.
11. The multiplexer passes the packet to the IP layer.

In the above manner, the link layer has single entry and exit points: the classifier and the multiplexer are visible to the IP layer, and the scheduler and demultiplexer are visible to the physical layer. Therefore, there is no need to modify the IP, MAC and physical layers. In addition, we can use protocol code which assumes that the protocol operates in isolation over the physical link. The main difference is that when multiple protocols operate concurrently, the round trip time of a packet is no longer fixed, due to unpredictable contention between the protocols.

In order to implement the MSLL framework we need three types of component:

1. Framework components, such as the classifier and the scheduler. This is the generic part of the link layer, reused over any wireless link and service.
2. Link layer protocols, such as ARQ and FEC. This is the protocol specific part, customized for specific wireless links and services.
3. Supplementary components, such as error models and applications. These are not part of MSLL, they are simply needed for performance testing.

2 Link layer protocols

Most protocols are included in the files `msll.h/cc`. The exceptions are the Snoop protocol, included in the files `mssnoop.h/cc`, as it is a port of existing code, and the RLC/AM protocol, included in the files `msrlcamhdr.h` and `msrlcam.h/cc`, as it was contributed by Michael Makidis.

2.1 Base protocol

The base MSLL class is the parent of all link layer protocols. An MSLL packet contains data and a header, struct hdr_msll. Since this is a simulator implementation, one header can contain the fields required by all protocols. For each specific protocol, only the fields that are actually used are counted when the packet is transmitted. The header includes header length, virtual time stamp, service number, frame type, block number, sequence number and acknowledgment number. The service number field is always sent and the time stamp field is never sent, while the rest depend on the protocol. The header length only needs to be sent when variable length headers are used.

The base MSLL class defines a Connector with two targets: Sendtarget_ points to the MAC layer and Recvtarget_ points to the IP layer. Packets are sent both ways since objects are bi-directional. All packets arrive at recv(), which looks at the header length field. If it is 0, the packet came from the IP layer and sendto() is called, else it came from the link and recvfrom() is called. While recv() is a generic dispatcher, the other two functions define the actual mechanism implemented; they are redefined by the actual protocols. In the base class, sendto() simply sets the service number and header length headers, keeps some statistics and calls recv() for Sendtarget_, while sendto() zeroes these fields, keeps some statistics and calls recv() for Recvtarget_.

The base MSLL class contains a service number and a header length variable, used to initialize each packet, as well as a function returning the protocol name in string format. While the class is called MSLL, so that all link layer modules can be descended from MSLL, it returns the string MSLL/Default, as this is actually the default (raw link) class. In OTCL we define a class MSLL/Default that inherits everything from MSLL and only create objects of that type. The class provides commands to return send and receive statistics, which are redefined by child classes, as well as commands to connect it with other objects.

2.2 FEC protocols

A *Forward Error Correction* (FEC) protocol transmits redundant information along with the data, so as to allow reconstruction of corrupted data at the receiver without feedback or retransmissions. This reduces delay but wastes bandwidth when there is either no need for recovery or when the redundancy included is insufficient for reconstruction. The protocols implemented here treat each packet as a unit which is either completely lost or correctly received.

The base MSLL/FEC class does not process packets, it simply adds a sequence number at the sender and counts losses at the receiver. Sequence numbers are 1 byte in size, so 1 byte is used for the header. Fixed packet sizes are used for simplicity. If a packet has a different size, it is forwarded as is. Since modules are bidirectional, they process both TCP data and TCP ACKs with different sizes. To allow each direction to use different sizes, we created TCP sinks that use their own flow ID, rather than echo the ID of the sender. This allows different modules for data and acknowledgments. The module provides a command to set the packet size.

The MSLL/FEC/XOR protocol sends one parity packet after every n (configurable) packets. The sequence number keeps counting and wraps around after 255, not after n. The parity packet is produced by XORing the n previous packets byte by byte, which requires considerable processing. If a single packet is lost out of a block, it is reconstructed by XORing the other $n-1$ packets and the parity packet. Each module keeps running sums for sent and received packets, both initialized to zeros. Each packet is XORed with the running sum, and at the end of the block what is left is the parity packet. This packet has invalid headers (the XOR of all previous headers), therefore its headers must be set just before transmission. Similarly, if a packet is reconstructed after reception, its headers are also invalid and must be set by the receiver.

The receiver counts losses and every time a block ends it checks whether lost packets can be recovered from (i.e. if exactly one packet was lost), otherwise it drops the parity packet. If multiple packets are lost, block boundaries may be crossed, so we have to start a new block by skipping over multiple losses. Reconstructed packets are delivered out of sequence to reduce delay, i.e. when a packet is reconstructed it is delivered after packets with lower sequence numbers that were used for its reconstruction. In sequence delivery would be useful for TCP, but complicated, as packets should be buffered after a loss and a timer would be needed to abandon recovery after some time.

If the sender stops sending before completing a block, no parity packet is produced, hence, if a packet is lost it cannot be reconstructed until the sender starts transmitting again and completes the block. To avoid this delay, the MSLL/FEC/XOR/Ad (adaptive) class starts a timer after sending each data (not parity) packet. If the timer expires before a new packet is sent, the parity packet is sent anyway, so reconstruction delay is kept low. This requires a timer which simply calls the proper function of the class.

With this scheme the block size varies, hence we need an end-of-block flag (1 bit) to signify block ends, as blocks may end after less than n packets. In addition, we need a block number (3 bits) to count blocks, so as to detect loss of complete blocks. The receiver uses the flag to see if a packet is parity, and the block number to check whether the parity belongs to the current block, or some blocks were completely skipped. In this case the sequence numbers are relative to the block number, i.e. each block starts counting packets from 0, instead of wrapping around after 255. Otherwise, this class is the same as MSLL/FEC/XOR.

Another option is the MSLL/FEC/RC class, which basically repeats the transmission of each packet n (configurable) times. This class is similar to MSLL/FEC and it only makes sense for extremely harsh environments.

2.3 ARQ protocols

An *Automatic Repeat reQuest* (ARQ) protocol uses retransmissions to recover from losses. This leads to high and variable delays, but relatively low overhead since redundant data are only transmitted when requested. The ARQ schemes implemented here are window based, where the *window* is a circular buffer of packets. When a packet is sent, it is stored at the next available position in the buffer. If the packet is not ACKed on time or if a NACK arrives, the packet is retransmitted. If the packet is ACKed, it is removed from the buffer. The buffer size n (configurable) determines how many packets may be awaiting acknowledgment at the sender, and must be large enough to allow continuous transmissions until ACKs are returned by the receiver, otherwise the sender will stall. At each point in time, some consecutive slots are used in the circular buffer by pending packets. This is the window that slides as new packets are transmitted and old ones are ACKed.

The base MSLL/ARQ class simply sends and receives packets and ACKs, without performing any error recovery. Each packet contains two sequence numbers: its own sequence number, incremented for every new packet, and the sequence number of the next expected packet, i.e. an ACK. This saves ACKs when there are regular packets to send. Sequence numbers uses 1 byte each in the header, while half a byte is used for the packet type (data or ACK), packed in 1 byte with the service number. To avoid waiting forever to send an ACK, after a packet is received a timer is (re)started. If no packets are sent until it expires, a regular ACK is sent. This is called a *delayed* ACK. A timer is also defined for retransmission timeouts, but it is unused in this class.

The MSLL/ARQ/GBN (Go Back N) protocol uses a real window of size n and makes retransmissions. When a packet is sent, a timer is started for it. If the packet is received in sequence, it is passed to the higher layer, and a delayed ACK is scheduled to be returned to the sender. If the packet is not received in sequence, i.e. some losses took place before it, it is dropped and no ACK is returned. Eventually, the sender times out for the lost packet (the oldest unACKed one), and all unACKed packets are retransmitted, since they must have been dropped by the receiver. When an ACK is received, either by itself or as part of a regular packet, the ACKed packet and all previous ones are released and their timers are cancelled, since they must all have been received. This class defines an array of pointers to buffered packets and an array of timers, one per packet. When a timer expires, it returns the number of the packet for which it was set. Note that ACKs refer to the next packet expected, not the last packet accepted by the receiver.

GBN wastes a lot of bandwidth as it retransmits all pending packets after each loss. MSLL/ARQ/SR (Selective Repeat) modifies the GBN scheme so as to only retransmit packets that were actually lost. This requires a buffer at the receiver to store all incoming packets. When packets are received out of sequence, they are stored at the proper position, with gaps left for missing packets. The first missing packet is negatively acknowledged (NACKed). A NACK means that everything before this packet was received, but the NACKed packet was not. Therefore, only one NACK is sent when multiple losses occur. When the first missing packet is received, another NACK may be sent for the next gap in the sequence. A NACK packet has the same format as an ACK but a different type.

The sender works as in GBN when sending packets. If the timer expires or a NACK is received for a packet, only this packet is retransmitted. When an ACK or a NACK arrives, all buffers corresponding to ACKed packets are released and their timers cancelled. At the receiver side, whenever a packet is received, all consecutive packets that are stored in the buffers are released to higher layers. Thus, if a missing packet arrives, all packets with higher sequence numbers that were already received are released along with it. Note that the sequence number space is $2n$ for n buffers, unlike GBN where the sequence number space is n for $n + 1$ buffers, hence care is needed to match packets with buffers. Two packet numbers correspond to each buffer, so we use modulo arithmetic to make the translation; on timeouts we look inside the actual buffered packet to see which one it is.

MSLL/ARQ/SR only sends one NACK at a time, so if multiple packets are missing the receiver does not send multiple NACKs until the first NACKed packet is received. In addition, when a NACK is lost we have to wait for the sender to timeout and retransmit the packet. The MSLL/ARQ/SR/Multi1 variant improves the situation by allowing each lost packet to be NACKed independently once (and only once), to speed up recovery. This means however that a NACK does not acknowledge all previous packets, since multiple packets may be NACKed at once. Hence, ACKs must always be sent whenever new packets are received, even if many NACKs are sent. When a NACK is lost, we still need to wait for a timeout. This variant is so close to the basic SR scheme, that only the receiver's code is modified to handle NACK transmission and reception. The trick is to keep a variable showing the highest sequence number seen at the receiver and never NACK any packets below that point, as they must already have been NACKed once.

The MSLL/ARQ/SR/Multi2 variant allows multiple NACKs to be sent for each missing packet, if needed. The sender is the same as in the basic SR scheme, but the receiver NACKs all missing packets when a packet is received out of sequence. The difference with the previous scheme is that we count the number of NACKs generated for each missing packet. If a packet that was NACKed arrives, then all packets with earlier sequence numbers that have been NACKed the same or a smaller number of times, must be NACKed again. This is because either their NACKs were lost or the packets were retransmitted but lost, as NACKs are sent always in sequence, therefore the retransmissions must also arrive in sequence. The receiver maintains an array showing how many times each packet has been NACKed.

Version 3 adds to all the Selective Repeat variants the capability to dynamically set their retransmission timeouts. When dynamic timeouts are used, each ACK is used to calculate a round trip time sample, and the samples are fed to a TCP style estimator of the average round trip time and its variance. The retransmission timeouts are then set using one of six possible TCP style formulas, that is, linear combinations of the average and the variance. Four more formulas represent slower and faster filtering versions of the two best performing TCP style formulas.

2.4 RLP protocols

The MSLL/ARQ/RLP protocol is Phil Karn's scheme for achieving mostly reliable transmission over cellular CDMA networks. It is a low overhead ARQ variant that differs from MSLL/ARQ/SR in two ways. First, ACKs are never sent, only NACKs are sent for missing packets, hence there is no ACK field in regular packets and only dedicated NACKs exist. Second, if a packet has been NACKed m (configurable) times and times out again, it is ignored and the receiver makes the loss visible to the IP layer. Note that in this protocol it is the receiver that times out after sending a NACK and not receiving the indicated packet. The goal is to avoid retransmitting the same data forever, since higher layers such as TCP will timeout and retransmit it anyway.

The sender in MSLL/ARQ/RLP simply buffers outgoing packets in case they must be retransmitted, but never releases them, since there are no ACKs. If the buffer is exhausted, it starts again at the beginning, dropping the packets already buffered. Note that the sender does not keep any retransmission timers, it only retransmits packets that are explicitly NACKed. This means however that when the sender has no data to send, the last packets sent may be lost and the receiver will never know, hence the sender will not notice that a loss has occurred for a long time. To avoid this, after every transmission a keepalive timer is (re)set, so that if an idle period starts, the timer will expire and a keepalive packet will be sent, containing the highest sequence number sent. Packets contain a 1 byte sequence number and a half byte packet type.

The protocol entity may receive three types of packet. If a NACK is received, the corresponding packet is retransmitted. If a data packet is received, the receiver's state is updated, and NACKs are sent for any missing packets; any packets received in sequence are released to higher layers. If a keepalive is received, the receiver's state is updated, and NACKs are sent for any missing packets.

When many packets have been lost in sequence, multiple NACKs are sent back to back and multiple timers are set at the receiver. To avoid the expiration of all these timeouts while the packets are still being received, a delay must be inserted between these timeouts. Therefore, we keep track of the last timeout set and we schedule a new timeout after either a full timeout period from now (i.e. a round trip time), or after a spacing period from the last timeout set (i.e. one packet transmission time) from the last timeout set, whichever is greater. For each new NACK a counter is set to 1.

Two timers may expire in this protocol. The keepalive timer expires during idle periods. It simply causes a keepalive packet to be sent and reschedules itself. The NACK timer expires if a NACKed packet has not been received on time. If we have not reached the NACK limit, we simply send a new NACK and increase the counter. If the limit has been reached, we release or drop all packets up to this one, and then release all packets after this one up to the next gap in the sequence. Packets should time out in sequence due to the way we schedule consecutive NACKs, but we always check from the beginning to make sure.

While MSLL/ARQ/RLP was designed for TCP where in sequence delivery and high reliability are critical, the protocol is fast enough to be used for real time UDP traffic too. By reducing the NACK limit to one, i.e. only one retransmission is allowed, the delay is reduced. However, when a packet is lost the packets received afterwards will need to wait for it to be received or dropped. By releasing packets as they arrive, rather than in sequence, we can further reduce delay. This is appropriate for UDP applications that either do not care about sequencing, or have their own resequencing buffers, such as audio/video applications that play back data at a constant rate.

This policy is implemented by MSLL/ARQ/RLP/OOS (out of sequence). It works exactly the same as MSLL/ARQ/RLP, but received packets are passed to higher layers immediately on reception. As a result, there is never any need to check which packets must be released to higher layers. Even further, there is no need to use a buffer at the receiver. Only an array of flags is needed (one bit per packet) to keep track of received packets. Only the receive and NACK timeout functions are modified to release all packets immediately and mark their arrival on the flags array to keep track of the current state.

2.5 SNOOP protocol

The Snoop implementation, MSLL/Snoop, is a port of the code included in ns-2. It is modified to send/receive packets via the MSLL interface instead of via an ns-2 LAN link, and its data structures were renamed with an MS prefix to avoid naming conflicts. Snoop only works at the base station or proxy, the mobile station does not need to do anything. This also means no retransmissions for data sent by the mobile station, unless if ELN is used by TCP.

Since this code is ported, it is quite different than the rest. It is split in the MSLL-Snoop and MSSnoop objects. MSLLSnoop is the main object for the link, and MSS-noop is the connection object dealing with a specific source-destination pair. Many connection objects are needed since this protocol uses TCP sequence numbers to maintain its state, hence many state machines are in operation in parallel. Snoop does not use fixed retransmission timers, it estimates link delay using a technique similar to that of TCP. If the integrate variable is set, the estimates are kept at the parent for all connections, else each connection handles its own estimates.

The only thing that the main object does is receive packets using the MSLL interface either from the wireless link or from a higher layer, locate the appropriate connection object and pass it the packet for further processing. When the connection object is done with the packet, it calls one of two helper functions in the main object which simply forward the packet in the appropriate direction. The connection object is allocated on demand and stored in an associative array (provided by OTCL), indexed by source/destination IP address (normally, the ports would also be needed to allow multiple sessions from the same IP address). The OTCL associatve arrays make the protocol easy to implement but very slow to simulate; a hash table would have been better.

In each connection object there are two buffers. The uplink (mobile to base station) uses a sequence number buffer to store the data from the packet headers received from the mobile station. The downlink (base station to wireless) uses a packet buffer, since packets may be retransmitted over the downlink. Instead of including state in a special packet header, as in the original code, this state has been moved to a separate structure, stored in an array that parallels the packet buffer array. LAN related state/code were removed and statistics were added.

For data packets arriving from higher layers, the protocol checks to see if the packet has been acknowledged or not. If not, the packet is either entered into the packet buffer in the correct position, or its data are updated if it was already there. The acknowledgments for these packets are handled by sending a retransmission if enough duplicate ACKs have been seen or by releasing some packet buffers if the ACK covers them. Duplicate ACKs for buffered packets are suppressed and all other ACKs are propagated. A timer is used for the earliest non ACKed packet; when it expires, the packet is retransmitted if it is the next one expected by the receiver, and the timer is rescheduled.

In the reverse direction, data from the wireless link are handled by keeping track of which packets have been sent in a buffer. Each entry holds a sequence number and the number of consecutive packets seen. The ACKs for these data packets are handled by looking for gaps in the sequence, and if one is found, by setting the ELN bit in the TCP header to inform the wireless host that a loss has occurred. If the TCP sender side is aware of ELN, it will retransmit the packet following the ACK. In `ns-link.tcl` the regular Snoop procedures are declared again for MSSnoop, omitting the redundant elements that are not part of MSSnoop.

2.6 RLC/AM protocol

The MSLL/ARQ/RLCAM protocol is the *Radio Link Control* (RLC) protocol in *Acknowledged Mode* (AM) defined by the *3rd Generation Partnership Project* (3GPP) for *Universal Mobile Telecommunication Networks* (UMTS). The protocol was implemented by Michael Makidis following the 3GPP Technical Specification 25.322 Version 7.0.0. Most features of the protocol are supported, including different ways of discarding persistently lost packets, multiple feedback options and various status reporting options and timers. Fragmentation and packing are not supported; the protocol turns network layer packets into link layer frames, regardless of their size.

3 Framework components

The classifier, the demultiplexer and the multiplexer are implemented in OTCL in files `ns-link.tcl` and `ns-lib.tcl` and in C++ in files `msllaux.h/cc`. The scheduler is implemented in files `msllscfq.h/cc`.

3.1 Packet handling

To guarantee that incoming IP packets have empty MSLL headers we use an MSHead object at the entry to a link to zero this size field. This is needed as in ns-2 all headers are present in each packet at all times. After a link layer service processes the packet it sets the header size appropriately before transmission. This enables the distinction of packets from the IP layer and packets from the physical layer.

Packets from the IP layer are directed to link layer protocols using any appropriate IP fields. In ns-2 ports are not well known, so we employ flow IDs set by the applications, hence we use a Classifier/Hash/Fid supplied by ns-2. The classifier uses a hash table to map these values to appropriate protocols. This table contains pointers to link layer protocols and the hashing function maps flow ID values to table entries. Many entries may point to the same link layer protocol. The first entry always points at the Default service. We only need to create and setup this element in OTCL.

The demultiplexer Classifier/MS/Demux looks at the MSLL header of a packet received from the link and passes the packet to the protocol corresponding to the protocol ID added by the scheduler before transmission. The multiplexer must simply pass the packets released from the link layer to the IP layer. Instead of defining a real multiplexer, all protocols simply point to the IP layer directly.

In order for packet transmission to take into account the additional overhead of the MSLL header, we define DelayLink/MSLL which is the same as DelayLink but using the total size of the packet to calculate the arrival time at the destination.

To keep most buffering at the IP level, we use a main IP level queue and a delay element before entering the link layer. To make the link layer seem busy for the nominal packet transmission time, we define DelayLink/Nominal which simply passes the packet downstream but only allows the upstream sender (the IP layer) to continue after the transmission would have completed with a single service.

The reason for the nominal delay is that the goal of the link layer scheduler is simply to protect each service from the overhead introduced by other services; it is supposed to follow the packet scheduling decisions made by the IP layer, not enforce its own. If we did not introduce this delay, when a fast wired link fed a slow wireless link, the IP level queue would pass many packets to the link layer scheduler, which could rearrange them based on its own scheduling policy.

3.2 Packet scheduling

We define an SCFQ scheduler by inheriting from the Queue object so as to be able to use it in place of a regular FIFO queue. The Queue/SCFQ object contains a number of FIFO queues (one for each link layer protocol, using the PacketQueue object defined by ns-2), an array with each element pointing at one queue, a heap to keep the queues sorted (actually, the timestamps of their first packets) which uses the Heap object defined by ns-2, and an array with the service rates (one entry per protocol). There are two types of SCFQ schedulers, Queue/SCFQ/IP for use at the IP level (uses the IP flow ID) and Queue/SCFQ/LL for use at the link layer level (uses the MSLL service number). In a real system, only the link layer scheduler is actually needed. The scheduler allows setting the rate for each service separately.

There are two functions used by the scheduler. Function deque() is called when the link becomes available in order to select the next packet for transmission. It extracts the queue whose first packet should be transmitted next (it is always at the top of the heap), causing the heap to be automatically resorted. Then the first packet of the queue is removed, the virtual time of the scheduler is updated, and the queue is inserted again into the heap, causing the heap to be automatically resorted. Finally, the packet is passed for transmission to the MAC layer.

Function recv()is called when a new packet is passed to the scheduler from a link layer protocol. First we find out the size of the packet, and then we locate its queue based on its protocol ID. If the queue is empty, we add the time needed to transmit the packet in an ideal fluid flow system (using the link shares table) to the virtual time of the scheduler and stamp the packet with this virtual time. Then we add the packet to the queue and insert it into the heap, since it is not empty any more. If the queue was not empty to begin with, we use the virtual time of the last packet in the queue as the basis for the virtual time calculation, and add the packet to the queue, without sorting the heap again. Finally, if the link was idle, which is the case when all queues are empty for some time, we transmit the packet.

It is also possible to use a FCFS scheduler at both levels, by simply using a standard Drop Tail queue rather than an FCFS queue. This option is useful for studying systems where multiple link layer sessions compete for the link without any scheduling. In this case the nominal delay element is not needed, but it also does not influence the results.

3.3 Multi service links

The multi service links are basically ns-2 links augmented with extra modules. They are implemented entirely in OTCL using the class MSLink which inherits the SimpleLink class of ns-2. In `ns-lib.tcl` we modified the simplex-link OTCL procedure to also create these links. The trick is to pass as the queue type either MSLL or NOMSLL and specify the details as additional arguments. MSLL means an MSLink with a SCFQ queue at the link layer level and any type of IP queue, while NOMSLL means an MSLink with a Drop Tail link layer level queue and any type of IP queue. In both cases, the remaining arguments specify the type of IP level queue needed and the number of services. This procedure creates the queues for the link layer and IP levels using the appropriate number of services for SCFQ queues and then the MSLink is created by passing the queues created and the number of services.

A link of the MSLink class is created using the init procedure which first creates a SimpleLink and then creates the additional elements, i.e. the head, the classifier, the demultiplexer and the nominal delay. Then the elements are connected in the order IP queue, nominal delay, head, classifier, link layer queue, all SimpleLink elements, demultiplexer. The set-modules procedure gets two lists of link layer modules and connects them into the link in sequence (in one direction), also inserting the modules in a table. The dump-modules procedure asks each module for statistics using this table to locate them. Finally, the set-services procedure gets a list of module names, creates the relevant modules and installs them in both directions between two nodes (calling set-modules twice). Note that the same module is installed in both directions, but each module has both send and receive capabilities. Services that only run on one side of the link are matched with a default service to make a pair.

To allow setting all rates at once, the set-rates procedure sets the rate table in one direction of the link. We do not set both directions at once because each traffic direction may carry a different mix of data. For example, downloading a file with FTP means a lot of packets on the downlink but a few acknowledgments on the uplink. The get-service and get-modules procedures return the services installed on an MSLink.

Since the simulator has generic procedures for all links, we must modify their behavior for an MSLink. The all-connectors procedure which applies an action to all connectors is redefined to avoid resetting the classifier, which would lead in its disconnection from its modules. The gen-map procedure generates a map of the objects in the simulation. Since MSLink objects have multiple paths, we define fake targets for the classifiers. These targets simply bypass the service modules and point to the next object. Finally, since only one queue and link are entered in the global arrays indexed by nodes, we use the IP level queue (not the link layer level queue) and the actual link delay (not the nominal delay) as representatives of an MSLink.

4 Supplementary components

The error models are in file `msllerr.h/cc` and the application agents are in files `msllagnt.h/cc` and `expoo.cc`. Some miscellaneous definitions and modifications are in files `queue.h` (Version 1), `ns-default.tcl` and `ns-packet.tcl`.

4.1 Error models

The new error models are variants and extensions of the basic error models of ns-2. ErrorModel/MS is exactly the same as ErrorModel, but it also includes the MSLL header in the packet size. Similarly, ErrorModel/Twostate/MS is the same as ErrorModel/Twostate, but it also includes the MSLL header.

In the simple two state model, packets are lost in the bad state with certainty. The variant ErrorModel/TwoState/MS/BER assumes that each state has a bit error rate (by default 0 and 1, i.e. good and bad) and it calculates the error probability for each packet using these rates depending on the current state. In the byte and time oriented cases a packet may cross multiple states, so all the probabilities must be factored into the final decision. In the time oriented case the intervals progress even when no packets are transmitted. A separate random variable can optionally be used for these calculations, allowing a private random number generator to be used for the error model.

A new model is ErrorModel/MS/Interval which uses a random variable to draw intervals between successive errors. In the packet case one packet is lost after each interval. In the byte case one packet is lost if the interval ends in a byte inside the packet. In the time case one packet is lost if the interval ends inside the packet (the error is instantaneous). In the time case the intervals progress even when no packets are transmitted. This model is the same as the one used in the Snoop papers.

The `ns-link.tcl` file also contains some modifications related to error models. The procedure trace-error is added to the Link class, allowing us to install a Trace/Loss object on the error model. The SimpleLink class keeps two new variables for the error model object and the trace object.

4.2 Application agents

The Application/Traffic/Exponential class in `expoo.cc` uses two exponential random variables (part of the application object) to generate on periods (in packets) and off periods (in seconds). Since these random variables use the default random number generator, their behavior depends on other objects. MSLL Version 1 adds the Application/Traffic/Exponential/MS class which uses pointers to external (exponential) random variables. The user can create these variables in the simulation script, associate them with a private random number generator and bind them to the application agent. In Version 2 the base class itself supports a command to associate a private random number generator with the internal exponential random variables, therefore there is no need to define a child class to do so.

The agents in `msllagnt.cc/h` are basically instrumented versions of the usual UDP and TCP source and sink agents. For UDP, the Agent/MSUDP class is a variation of Agent/UDP, but each packet is numbered using the RTP sequence number field and carries a default timestamp instead of an RTP timestamp. The Agent/MSSink class is a matching instrumented sink; it counts packets and bytes received, out of sequence packets and stores delay samples for each packet in a Samples object in order to provide delay statistics. This agent also tracks the highest packet number seen and the time of arrival of the latest packet.

For TCP Agent/TCP (Tahoe), Agent/TCP/Reno or Agent/TCP/Newreno can be used as the source with one of two instrumented sinks, Agent/TCPSink/MS for regular ACKs and Agent/TCPSink/DelAck/MS for delayed ACKs. The sinks are the same as the regular ones but they track the number of received packets and bytes, the delays in Samples objects and the latest arrival. In addition, if flowid_ is not -1 they use their own flow ID rather than echoing the flow ID of the source. The use of a different flow ID allows ACKs to use a FEC module appropriate for their packet size. Version 3 adds two sinks for the Agent/TCP/Sack1 (SACK) variant of TCP: Agent/TCPSink/Sack1/MS for regular and Agent/TCPSink/Sack1/DelAck/MS for delayed ACKs.

For HTTP there is an application level statistics agent called Agent/MSHTTP. This agent is neither a source nor a sink for HTTP data, it is a sink for statistics only. At the conclusion of each transaction, the HTTP objects pass their statistics (busy/idle time, number of requests to the server and number of packets from the server) to this agent as a string by using the record command.

Since Version 4, all the above sinks accept the resetstats command which resets all statistics. The UDP and TCP sinks also accept the mark command which causes all current statistics to be saved, so that they may be displayed along with the latest statistics, when requested. The HTTP sink does not accept the mark command, as it only saves statistics at the conclusion of a transaction. In multiple application tests, every time an HTTP transaction finishes, a mark command is issued to the FTP and CBR agents to synchronize their statistics with those of HTTP. Since the mark command does not reset the statistics, it does not matter how many times it is issued, it will always record the statistics since the last resetstats command was issued.

4.3 Miscellaneous

The only change needed in the `Makefile` is to add the MSLL object files: `msll.o`, `msllaux.o`, `msllscfq.o`, `msllagnt.o`, `msllerr.o` and `mssnoop.o`; since Version 4, `msrlcam.o` also needs to be added. All other changes are in C++ or OTCL files that are already required to build the simulator. In Version 1 all C++ files are in the main ns-2 directory, while in Version 2 and later they are in the subdirectory `msll`; note that in Version 2 and later there are no changes to other C++ files. In Version 2 and later we can set the OLD_RNG compiler flag to use the old random number generator for results comparable to the original version. However, differences in the implementation of the simulator mean that the results are not exactly the same. In Version 3 the results are further different due to the nearly exclusive use of the default options for TCP. Since Version 4 the code has been modified to work better with the new random number generator, therefore results again differ from previous versions.

In Version 1 a change is made to `queue.h`: we add a real tail pointer called ttail_ to the class PacketQueue which is part of all actual queues. This pointer points at the last packet in the queue or to nil if the queue is empty. It is used by the SCFQ scheduler (`msllscfq.h/cc`) to access the timestamp of the last packet in the queue when it needs to calculate the timestamp of the next packet to add to the queue. In Version 2 amd later there is no need to change `queue.h` as the tail_ pointer now actually points to the last packet, hence `msllsfq.h/cc` refer to the existing pointer.

In `ns-default.tcl` default values are set for those variables that are visible from OTCL, such as timer durations and retransmission limits. Most of these values are set in the simulation scripts to their proper values. Finally, in `ns-packet.tcl` we define a header named MSLL which is simply a placeholder for the actual MSLL header in the packet objects.

5 Supplementary scripts

5.1 Execution scripts

The execution scripts are included in the `runscr.tgz` file. The main simulation script is `skeleton.tcl` which sets up the simulation topology and performs a single experiment. The remaining scripts are simply driver files used to iteratively call `skeleton.tcl` with a range of parameters (e.g. multiple runs for specific combinations of link type, topology, error rate and application mix). For example, `rlp.tcl` executes multiple simulations with ARQ/RLP used for the TCP based service.

The `skeleton.tcl` script is ran as `ns skeleton.tcl <parameters>`. Without any parameters it prints out its various options. The first two parameters, which are mandatory, are the filename where output is sent (stdout for output to the screen) and the run number. The filename is used to create a file when the run number is one and to append to this file when the run number is greater than one; this allows sending multiple runs to the same file. The run number selects a predefined good seed for the ns-2 random number generator, which is used for the error models. The HTTP and CBR applications use private random number generators always seeded with the same seed, to avoid changing their behavior when the error behavior changes.

If the options to produce trace files are set, these files have the same name as the output file with a suffix depending on trace type. Packet traces have the suffix `.tr`, nam traces use `.ntr`, error traces use `.etr` and TCP traces use `.ttr`. By default, the output directory is `../output`, so if you run your script from `$HOME/ns/runscr`, then the output will go to `$HOME/ns/output`.

The main options of the `skeleton.tcl` script are the topology (one LAN or WAN wired link and one or two wireless links, called LAN1, LAN2, WAN1 and WAN2), the wireless link type (Cellular, PCS, WLAN, HSCSD or BLAN), the error level (0 to 4, 0 is no errors) and the application mix (FTP only, HTTP only, CBR only, all applications with scheduling under SCFQ, FTP/CBR with scheduling under TWO, HTTP/CBR with scheduling under TWOH, all applications without scheduling under ALL, FTP/CBR without scheduling under AFC, HTTP/CBR without scheduling under AHC, FTP/HTTP without scheduling under AFH (obsolete), FTP/CBR with CBR at the wired link only under CAFC, HTTP/CBR with CBR at the wired link only under CAHC). A primary link layer type can be specified when a single service is used; in multiple service tests, the primary link layer type is used for the TCP service, and the secondary link layer type is used for the UDP based service.

The script starts by instantiating the simulator and reading any options. If not enough parameters are passed, a synopsis of the parameters is printed. Then the random seed is initialized from a predefined table, any required trace files are opened and the WWW model is initialized, i.e. the distribution functions are loaded, a private random number generator is initialized and HTTP options are set. Then the topology is setup using whatever parameters were defined, the error modules are created and installed at each wireless link, the application and wireless link modules are created and initialized, the service (link layer) modules are installed in each link and the application agents are created and installed at the appropriate nodes. The default parameters for each link type, such as speed, delay, error model and error level parameters, are printed; the same parameters are always printed to simplify parsing the output, although only some of them used by each test. Finally, the simulation is started by asking the application modules to start, and an event is scheduled to end the simulation, either at the end of an FTP transfer or at a predefined instant. When the simulation ends, results are printed from each application module and each link; the exact output depends on the application and link layer modules used.

The scripts calling `skeleton.tcl` use the following scheme for the output files:

Topology (2): L/W for LAN/WAN, 1/2 for one/two wireless links.

Link type (1): C/P/W/H/B for Cellular, PCS, WLAN, HSCSD, WLAN.

Application mix (1): The values used are:

- C: CBR
- F: FTP
- H: HTTP
- S: SCFQ (FTP, HTTP and CBR with scheduling)
- A: ALL (FTP, HTTP and CBR without scheduling)
- T: TWO (FTP and CBR with scheduling)
- U: TWOH (HTTP and CBR with scheduling)
- V: CAFC (FTP and CBR with CBR at the wired link only)
- W: CAHC (HTTP and CBR with CBR at the wired link only)
- X: AFC (FTP and CBR without scheduling)
- Y: AHC (HTTP and CBR without scheduling)
- Z: AFH (FTP and HTTP without scheduling, obsolete)

Error level (1): 0/1/2/3/4

Link layer (2): Primary (TCP when two services are used) first, secondary (UDP when two service are used) next; if one service is used, the secondary is ignored. The values used are:

- 0: Default
- 1: FEC/XOR
- 2: FEC/XOR/Ad
- 3: ARQ/GBN

- 4: ARQ/SR
- 5: ARQ/SR/Multi1
- 6: ARQ/SR/Multi2
- 7: ARQ/RLP
- 8: ARQ/RLP/OOS
- 9: Snoop
- A: RLC/AM

For example, `L1CF130` means a wired LAN with one Cellular wireless link, FTP only, error level 1 and `ARQ/GBN` used for TCP.

Since Version 3 any additional options are indicated by an additional letter or number in the end of the filename. For the dynamic timeout option of the selective repeat protocols, the last letter used is:

- 0: rtxto = 1+4 (dynamic)
- 1: rtxto = 2+4 (dynamic)
- 2: rtxto = 3+2 (dynamic)
- 3: rtxto = 4+0 (dynamic)
- 4: rtxto = 4+2 (dynamic)
- A: rtxto = 0.9 (static)
- B: rtxto = 1.0 (static)
- C: rtxto = 1.1 (static)
- D: rtxto = 1.2 (static)
- E: rtxto = 1.3 (static)

The following options were removed in Version 4:

- 5: rtxto = 4+4 (dynamic) - removed in Version 4
- 6: rtxto = 3+2 fast (dynamic) - removed in Version 4
- 7: rtxto = 3+2 slow (dynamic) - removed in Version 4
- 8: rtxto = 4+0 fast (dynamic) - removed in Version 4
- 9: rtxto = 4+0 slow (dynamic) - removed in Version 4

Additional scripts were added to generate these files, named with the suffic `-dyn`. They simply pass the additional arguments to the simulator as a string and name the files with an extra letter at the end.

Since Version 4 tests with TCP Reno and TCP Sack were added, indicated by an additional letter in the end of the filename. For such tests the last letter used is:

- R: TCP Reno
- S: TCP Sack

Additional scripts were added to generate these files, named with the suffic `-sack`. Since Version 5, TCP Sack is the default TCP variant.

Since Version 4 tests with RLC/AM were added, indicated by an additional number in the end of the filename. For such tests the last number used is:

- 3: MaxDAT = 3 (discard a frame after 2 transmission attempts)
- 4: MaxDAT = 4 (discard a frame after 3 transmission attempts)
- 5: MaxDAT = 5 (discard a frame after 4 transmission attempts)
- 6: MaxDAT = 6 (discard a frame after 5 transmission attempts)
- 7: MaxDAT = 7 (discard a frame after 6 transmission attempts)

A single script generates all these files.

5.2 Post processing scripts

The post processing scripts are included in the `procscr.tgz` file; each post processing step consists of a TCL script that calls an AWK script in a loop, so as to process a large number of output files. If your the files are located at `$HOME/ns/output`, then the scripts must be at `$HOME/ns/procscr`. The scripts output data to stdout, but they are usually redirected to `$HOME/ns/oldraw`. To convert the error levels in these data from the 0-4 range to actual frame error rates, the `conv.tcl` script is used; it takes two parameters and calls the `conv.awk` script in order to convert all result files from the directory passed as the first parameter, usually `$HOME/ns/oldraw`, to result files in the directory passed as the second parameter, usually `$HOME/ns/raw`. This simplifies graph generation, as the error rate can be directly used.

The `cbr.awk` script reads a file with multiple tests and produces average, minimum and maximum metrics for the loss rate (frame error rate), average packet delay, average delay plus and minus one and two standard deviations (seconds) and average loss plus and minus one and two standard deviations, for the CBR stream only. To calculate the delay plus deviation metrics, we compute the metric for each run separately and then average all metrics. The `cbr.tcl` script calls the `cbr.awk` script for all topologies, error levels and link types tested with a specific protocol and outputs one line per combination.

The `http.awk` script reads a file with results from multiple tests and produces average, minimum and maximum transactions per second, average, minimum and maximum throughput in Kbps and average throughput plus and minus one and two standard deviations for HTTP traffic. Throughput is server originated packets per unit time multiplied by packet size. We calculate these metrics separately for each test before averaging, using the time the last complete transaction ended, as opposed to total test time, so that transactions in progress are ignored both in terms of data and time taken. The `http.tcl` script is similar to `cbr.tcl`.

The `ftpt.awk` script reads a file with multiple tests and produces average, minimum and maximum throughput based on total transfer time and data size, as well as average throughput plus and minus one and two standard deviations. The `ftpt.tcl` script is similar to `cbr.tcl`. Finally, the `ftp.awk` script is the same as `ftpt.awk` but it only works for FTP tests as it calculates goodput for both wireless links in both directions (for one wireless link tests, the second pair of goodput values is zero). Goodput is calculated as total FTP file transfer size (not TCP data transferred, this includes duplicates) divided by total data sent on each wireless link. For ACKs, we use the number of TCP packets multiplied by 40. For delayed ACKs this should be divided by 2. The `ftp.tcl` script is similar to `cbr.tcl`.

Version 3 simplifies and standardizes the results in two ways. First, the `ftp.tcl` script is replaced with the `ftpt.tcl` script, that is, no goodput is calculated, therefore the results produced for FTP for all application mixes are the same. Second, all scripts output the 95% and 99% confidence intervals of each metric after its average, minimum and maximum value, replacing the average plus/minus one/two standard deviations. A few scripts, named with the suffic `-dyn`, were also added to process the dynamic timeout result files; they use the last letter of the filename to insert a new column describing the timeout option used just before the error level. Since Version 4, a few scripts, named with the suffix `-sack`, were added to process the Reno and Sack result files; they use the last letter of the filename to insert a new column describing the TCP variant used just before the error level. A few scripts, named with the suffix `-rlc`, were added to process the RLC/AM result files; they use the last number of the filename to insert a new column describing the MaxDAT parameter used just before the error level. The conversion scripts have also been modified to detect automatically whether the error level is in its standard column (3rd) or in the next one (4th).

6 Version History

- 1999: Original version (not available).
- 2000: Minor extensions and corrections (not available).
  - Added error level 0 (no errors).
  - Conversion of error levels to error rates for figures.
  - Fixed zero/sign bugs in processing scripts.
- 2002: Major extensions and corrections (Version 1).
  - Added HSCSD and BLAN link types.
  - Added mark command to CBR and FTP modules.
  - Changed the definition of ALL (two services without SCFQ, not one).
  - Fixed name clashes (changed xor() to msxor()).
  - Fixed HTTP direction bug (server and client were interchanged).
  - Fixed HTTP transfer size bug (client requests were also counted).

- 2004: Port to ns-2.27 (Version 2)
  - No changes needed to `queue.h` and `expoo.cc`.
  - Fixed FEC/XOR/Ad timer bug (the timer was not stopped on parity).
  - All C++ files were moved to the `msll` subdirectory.
- 2005: Dynamic timeouts (Version 3)
  - Added dynamic timeouts to SR protocols.
  - Only SCFQ and TWO use the SCFQ scheduler.
  - Added the AFC, AHC and AFH application mixes.
  - Modified processing scripts to return confidence intervals.
  - Added sinks for SACK TCP.
  - Redefined the characteristics of Cellular and PCS links.
  - Redefined the CBR rate for Cellular and PCS links.
- 2007: Port to ns-2.30 (Version 4)
  - Added the UMTS RLC/AM protocol (written by Michael Makidis).
  - Added the CAFC and CAHC application mixes (contention on wired link).
  - Added the TWOH application mix (same as TWO but with HTTP).
  - Removed the AFH application mix.
  - Removed some of the dynamic timeout options (SR protocols).
  - Modified the dynamic timeouts (SR protocols) to be nondecreasing.
  - Added a reset command to all sinks (UDP, TCP, HTTP).
  - Fixed a division by zero bug in UDP sinks.
  - Improved random behavior with the new RNG (now standard).
- 2007: TCP like dynamic timeouts (Version 5)
  - The dynamic timeout estimators use the (integer) TCP code.
  - The default TCP variant is TCP Sack.

7 Installation

Version 1 works with ns-2.1b4, Versions 2 and 3 work with ns-2.27 and Versions 4 and 5 work with ns-2.30. In all cases the proper `ns-allinone` version must first be unpacked, for example under `/usr/local/src/`, configured and installed.

For Version 1, extract the files from `msll.tgz` in the ns-2 directory. Overwrite existing files when asked. Add the following object files to the `Makefile`:

- `msll.o`

File	**V1**	**V2**	**V3**	**V4**	**V5**
msll.h	1.22	1.23	1.25	1.28	1.29
msll.cc	1.25	1.26	1.33	1.36	1.37
msllagnt.h	1.8	1.10	1.12	1.16	1.17
msllagnt.cc	1.11	1.13	1.16	1.22	1.23
msllaux.h	1.3	1.5	1.6	1.7	1.8
msllaux.cc	1.3	1.5	1.6	1.7	1.8
msllerr.h	1.5	1.6	1.7	1.8	1.9
msllerr.cc	1.5	1.6	1.7	1.8	1.9
msllscfq.h	1.5	1.7	1.8	1.9	1.10
msllscfq.cc	1.5	1.7	1.8	1.9	1.10
mssnoop.h	1.3	1.5	1.6	1.7	1.8
mssnoop.cc	1.3	1.5	1.6	1.7	1.8
msrlcamhdr.h				1.1	1.2
msrlcam.h				1.1	1.2
msrlcam.cc				1.3	1.4
ns-default.tcl	1.10	1.15	1.16	1.20	1.21
ns-lib.tcl	1.7	1.10	1.11	1.13	1.14
ns-link.tcl	1.20	1.23	1.24	1.25	1.26
ns-packet.tcl	1.3	1.6	1.7	1.8	1.9

Table 1: File versions for each MSLL Version

- `msllaux.o`
- `msllagnt.o`
- `msllerr.o`
- `mssnoop.o`
- `msllscfq.o`.

Finally, do `make clean` and `make`.

For Version 2 and later, extract the files from `msll.tgz` in the ns-2 directory. Overwrite existing files when asked. Add the above object files to the `Makefile` with the prefix `msll/`. For Version 4 and later, also add `msrlcam.o`. For rough compatibility with Version 1, add `-DOLD_RNG` to the `Makefile`. Finally, do `make clean` and `make`.