Generic Multicast Transport Services: Prototype Design and Implementation CMPSCI TR 00-53

Jonathan K. Shapiro Don Towsley
Brad Cain
Department of Computer Science
University of Massachusetts at Amherst
{jshapiro, towsley}@cs.umass.edu

November 9, 2000

Abstract

The difficulty of implementing scalable multicast transport protocols under the standard IP multicast service model has engendered proposals to address the problem inside the network by deploying advanced services in routers. Generic Multicast Transport Services (GMTS), proposed by Cain and Towsley, is one example of this approach. GMTS is a set of simple filtering and subcasting services for source-based multicast, which can be used by protocol designers to improve scalability by supressing redundant feedback and isolating transmissions to subsets of a multicast group. For such services to be deployable in the internet, it must be possible to implement them simply in real routers with a minimal performance impact on existing traffic. To understand potential implementation issues, we have constructed a prototype GMTS router. This work presents a description of the implementation along with a preliminary performance evaluation aimed at understanding the overhead providing such services imposes on the router, itself. We also present some lessons learned and offer recommendations for addressing implmentation concerns in future versions of the GMTS specifications.

1 Introduction

The development of scalable end-to-end multicast protocols poses a tremendous challenge to network protocol designers. For example, the development of reliable multicast protocols has received considerable attention in recent years. Many protocols based on an end-to-end solution [4, 7, 9] have encountered problems scaling to hundreds, much less thousands, of

receivers. The primary obstacles to the development of scalable protocols have been *feedback implosion* and *transmission isolation*. The first of these concerns the difficulty for a large multicast application to limit feedback from receivers to a data source or to each other. The second concerns the difficulty of limiting the transmission of data to the subset of a multicast group that requires it. This type of restricted transmission is known as *subcasting*.

There have been several proposals for adding functionality to routers for the purpose of improving the performance of multicast applications, particularly reliable multicast. Papadopoulous and Parulkar [8] introduced additional forwarding functionality to a router which would allow each router to identify a special outgoing interface over which to transmit a particular class of packets. They showed that this *turning point functionality* could be used to improve the performance of reliable multicast protocols. Levine and Garcia-Luna-Aceves [6] proposed the addition of *routing labels* to routing tables, which could be used to direct packets over specific interfaces. One of these, called a *distance label,* was shown to be quite useful in reliable multicast for directing requests for repairs to nearby repair servers. The third and, perhaps most relevant proposal is the PGM protocol [3]. PGM is a reliable multicast protocol which uses negative acknowledgements (NAKs). The PGM protocol is an end-to-end transport protocol that contains a router component to perform NAK suppression and provide retransmission subcasting functionality. GMTS is especially motivated by PGM and the recognition of utility in exporting a set of flexible, simple router-based functionality (such as was used to implement PGM) for the purpose of protocol design. Such router support greatly simplifies the design of a large class of scalable multicast transport protocols. Recognizing the general usefulness of the PGM NAK suppression and subcasting mechansims, Yano and McCanne have recently advocated a generalization that integrates such mechansisms into the IP multicast service model [10].

In previous work, Cain and Towsley [1, 2] introduced Generic Multicast Transport Services (GMTS) to help protocol designers deal with scalability problems. GMTS generalizes many of the ideas contained in PGM, making a set of generic services available to support a wide range of multicast transport protocols. These services are designed to assist the large-scale collection of receiver feedback information and provide subcasting services for large multicast groups. They consist of simple filtering and aggregation functions residing within

routers and can be invoked from the edges of the network. GMTS services are implemented at the IP layer and provide unreliable *best effort* services. Transport protocols that make use of GMTS must be robust in the face of failures and in the absence of GMTS-capable routers in the network.

Since GMTS is not intended to provide sophisticated services which are difficult or impossible to implement in routers, our present work is aimed at exploring some of the implementation issues and demonstrating how GMTS services can be introduced in a router. To this end, we have constructed a prototype GMTS router and implemented a subset of the proposed GMTS services. In this paper, we describe the prototype design, evaluate its performance at modest loads, and review some of the lessons learned in its implementation.

The rest of this paper is organized as follows. In Section 2, we review the GMTS specification, paying particular attention to those features we have implemented in the prototype. We present the prototype design in Section 3, including a description of a daemon that provides the bulk of GMTS functionality and a set of minimal kernel modifications necessary to support the daemon. Sections 4 and 5 describe the implementation of a simple reliable multicast protocol and presents a limited evaluation of performance primarily aimed at establishing the correctness of the protocol and measuring the overhead required for GMTS processing at the router. We conclude in Section 6 by reviewing the lessons learned in the process of implementing the prototype and suggesting a few minor modifications to the GMTS specification.

2 Generic Multicast Transport Services

The services provided by GMTS are designed to operate over the unreliable IP multicast protocol. A GMTS transport session is built on top of a multicast session between a single source and multiple receivers. In addition to maintaining the routing state ordinarily associated the session, GMTS-capable routers on the multicast tree also maintain a set of GMTS objects. The session source is responsible for setting up these objects and periodically refreshing their state. Object methods can be invoked by either sender or receiver as needed to implement transport protocols. The GMTS specification includes a signaling protocol

to construct and invoke methods on objects in the network, and a collection of predefined object types.[1]

The GMTS signaling protocol is used by hosts to set up and invoke the provided services. A session source first initializes one or more desired services by sending a GMTS setup message to the multicast group address. GMTS-capable routers on the tree then aggregate feedback from receivers and isolate transmissions through the use of filters set by either the sender or the receivers. For robustness, the source periodically retransmits the setup message to refresh GMTS state in the face of routing changes and other possible errors. It should be stressed that GMTS services are only invoked for these signaling packets, which account for a small fraction of the total session traffic. Data packets are treated no differently from ordinary multicast traffic.

It is worth emphasizing that, while GMTS objects may be created within the network at the discretion of the session source, the specific functionality of GMTS objects is defined by the GMTS specification. GMTS does not permit the execution of arbitrary user-defined functions uploaded to routers. At present, only one object type has been defined. As has been shown elsewhere [1] this so-called *General Purpose Object (GPO)* can be used quite flexibly to implement a variety of multicast transport protocols.

The job of a GMTS-capable router is to process the signaling protocol and implement the functionality of of the objects defined by the GMTS specification. In Section 3 we will describe how we implemented a prototype GMTS router. Before we present our prototype implementation, however, we will present the GMTS signaling protocol and General Purpose Object specifications in more detail.

2.1 GMTS Protocol

The GMTS signaling protocol introduces two special packet types: A *state setup packet (SSP)* for declaring and initializing objects, and a *method invocation packet (MIP)* for remotely executing methods on objects. These signaling packets are trapped by GMTS routers

[1] Although GMTS currently only defines a single, highly flexible, object type, the protocol is capable of handling multiple object types as more services are proposed.

for special processing.

An SSP is multicast by the sender to the multicast group. As it traverses the multicast distribution tree, it is trapped by all GMTS-capable routers. Each router creates objects corresponding to the declaration within the SSP if the objects do not yet exist, and then forwards the packet. The second purpose of the SSP is to inform a GMTS router of the address of the next upstream GMTS router. This information enables GMTS sessions to send signals along the reverse multicast routing path via unicast, bypassing intervening routers that do not support GMTS.

To maintain robustness in the presence of changes in the multicast tree topology, the sender periodically rebroadcasts SSPs. GMTS routers that were on the original tree use these packets to refresh the state of existing objects, and new objects are created along new branches. A soft state mechanism controls object lifetimes so that objects residing on pruned branches eventually time out and are removed by their routers.

GMTS supports both reliable and unreliable method invocation through the use of MIPs. Each MIP identifies the methods to be invoked along with any required parameters. In addition, a MIP may include data. MIPs are primarily used by receivers to invoke object methods. These methods typically alter the state of the objects or perform feedback aggregation. Senders may also invoke methods. Each MIP contains one or more method invocations and may invoke methods on multiple objects. Methods are invoked in the order in which they appear in the packet. The presence of multiple methods in a packet does not imply a guarantee of atomicity—the methods are invoked as if each arrived in a separate packet.

To get an intuition of how methods may be invoked to implement a protocol, consider a very simple reliable multicast mechanism. Receivers might invoke a method to signal a negative acknowledgment (NAK) during a reliable multicast session. A sender would invoke a different method to send the corresponding repair packet. The first method propagates toward the source altering the state of objects in its path. The second method propagates down the tree along the paths defined by the object state, resulting in a repair packet arriving at each requesting receiver. We will return to this example throughout this discussion, filling in details to illustrate exactly how GMTS makes such a protocol possible.

2.2 GMTS Objects

The intention behind GMTS is to define a small set of specific object types that are sufficiently general to support a wide range of transport protocols and whose functions can be easily implemented in routers. The current GMTS specification defines a single object type,the *General Purpose Object (GPO)*, which provides basic feedback suppression and subcasting services. The GPO methods and data members are well-documented elsewhere [1] and we will not describe them completely here. However, we will review those aspects that have been implemented in the prototype.

The GPO is designed to support protocols that employ a sequence number space and thus maintains a set of data indexed by sequence number. Apart from a few parameters stored at the object level, virtually all of the storage used by the GPO is associated with this set of data. The object keeps data only for sequence numbers within a specified window and allocates storage sparsely as needed within this range. The sequence data includes a single non-negative integer, known as the *sequence state* and a vector of integers representing the interfaces of the router, which we call the *interface state vector*. A "soft state" timer is associated with each sequence number to limit the lifetime of sequence number data. Timers for sequence number data are typically set to expire much faster than the owning object and, unlike object-level timers, are not refreshed when an SSP is retransmitted.

Most of the GPO methods take a sequence number as a parameter and perform some operation on the associated sequence state and interface state vector. We have implemented two such methods.

The method $rcvr\_update(n, value, pred, f_s, f_v)$ performs feedback suppression and updates state at the routers where it is invoked. It takes as parameters a sequence number n, a non-negative integer $value$, a predicate identifier $pred$, and the identifiers of two update functions f_s and f_v, which update the sequence state and interface state vector, respectively. The method first evaluates the predicate $pred(value, state(n))$. If the predicate evaluates to true, then the update functions are applied to sequence state and interface state.

$$state(n) \leftarrow f_s(value, state(n)) \quad (1)$$

$$interface\_state(n) \leftarrow f_v(value, interface\_state(n), arrival\_vec), \quad (2)$$

$pred(val, s)$	$f_s(val, s)$	$f_v(val, ifs, arr)$	$g_s(val, s)$	$g_v(val, ifs)$
$val < s$	$val + 1$	$ifs \;\&\&\; arr$	$val - 1$	ifs
$val > s$	val	$ifs \;||\; arr$	val	$max(0, ifs - val)$
$val \leq s$		$val * arr$		$max(0, ifs - 1)$
$val \geq s$		$max(ifs, val * arr)$		
$val == s$				

Table 1: Predefined functions used by GPO methods. val and s are non-negative integers. ifs and arr are vectors of integers.

where *arrival_vec* is a vector of all zeros except for a one in the position corresponding to the interface over which the method was invoked. If $state(n)$ has changed, then the *rcvr_update* method is invoked over the link directed toward the source with $v = state(n)$ and all other arguments unchanged. Otherwise, no additional invocation occurs. This method is normally invoked by receivers.

Subcast operations are implemented by invoking the method $forward(n, value, g_s, g_v, data)$. When this method is invoked at a router, it is re-invoked on all interfaces (except for the one that the original invocation arrived on) for which the corresponding elements in $interface\_state(n)$ are greater than *value*. In addition, the sequence state and interface state are updated according to the update functions g_s and g_v.

$$state(n) \leftarrow g_s(value, state(n)) \tag{3}$$

$$interface\_state(n) \leftarrow g_v(value, interface\_state(n)), \tag{4}$$

This method is typically invoked by the sender.

In the methods presented above, the values of *pred*, f_s, f_v, g_s are selected from sets of predefined functions. Table 1 lists the available functions for each argument. In Section 5 we will demonstrate how these two methods can be used to implement a simple reliable multicast protocol.

2.3 A Simple Protocol

Let us now return to our previous example and consider in more detail how services provided by the GPO can be used to implement a simple reliable multicast protocol. More sophisti-

cated protocols using GMTS are described in [1], but this simple one will be adequate for the purposes of illustration.

Initialization: The source periodically multicasts an SSP that declares a GPO with an associated timeout. The SSP initializes variables in the object for the start and end of the sequence number window, and default sequence number state values and their associated timeouts. As the session proceeds, the source may also use these periodic SSPs to advance the window by reinitializing certain variables. Routers along the multicast tree construct the object, record the address of their nearest upstream GMTS router and overwrite this field of the GMTS header with their own address as they forward the SSP. Receivers record the address of their nearest upstream GMTS router and the current sequence window.

Data transfer: Data packets are multicast to the group. Each packet contains a unique sequence number. Receivers use these numbers to identify lost packets. These packets require no GMTS support and are treated by the router as ordinary multicast packets.

Repair requests: When a receiver believes it has lost a packet with sequence number n, it invokes the *rcvr_update* method with $seq = n, value = 1, pred =>, f_s(x, y) = x$, and $f_v(value, v_1, v_2) = \max(v_1, value * v_2)$ by sending a MIP via unicast to its nearest upstream GMTS router. The operation of the *rcvr_update* method causes the router to maintain an interface vector for sequence number n that contains a 1 for each interface on which a NAK is received. Furthermore, the router forwards the first such NAK towards the source but suppresses subsequent NAKs.

Repair transmission: When the sender receives a copy of the MIP described above, it interprets it as a repair request and immediately constructs a repair packet. The repair packet is, itself, a MIP invoking the $forward$ method with $seq = n, value = 1, g_s = x$, and $g_v = v$. The *data* parameter contains the contents of the lost packet. The operation of $forward$ at the router reinvokes the method only along those interfaces which were marked as the result of receiving a repair request. This MIP is eventually received by those (and only those) receivers who requested a repair, where it is interpreted as a repair packet and its data extracted.

In Section 3 we will describe our prototype router and revisit this example once more when we describe our prototype router to see exactly how the router implements the actions

described above. We will also see that adding the necessary functionality to a real router introduces a modest overhead, which is tolerable for many applications.

3 Router Prototype Design

We designed the GMTS prototype with several objectives in mind. First, we wanted to validate the GMTS protocol specification by implementing a significant portion of it. Second, we wanted to create a prototype that could be easily extended over time. Third, we wanted to create a prototype that would run on a standard Linux PC configured as a router.

To be minimally functional, a GMTS router must have several features. It must be able to process SSP and MIP packets as they traverse the router to create GMTS objects and invoke methods on those objects. A timeout mechanism is required to handle object-level timeouts and other scheduled operations. The router must intercept multicast GMTS packets and modify their NEXT_HOP_ADDRESS field before forwarding them. Finally, the router must be able to perform a subcast operation, forwarding a multicast packet on a subset of downstream interfaces.

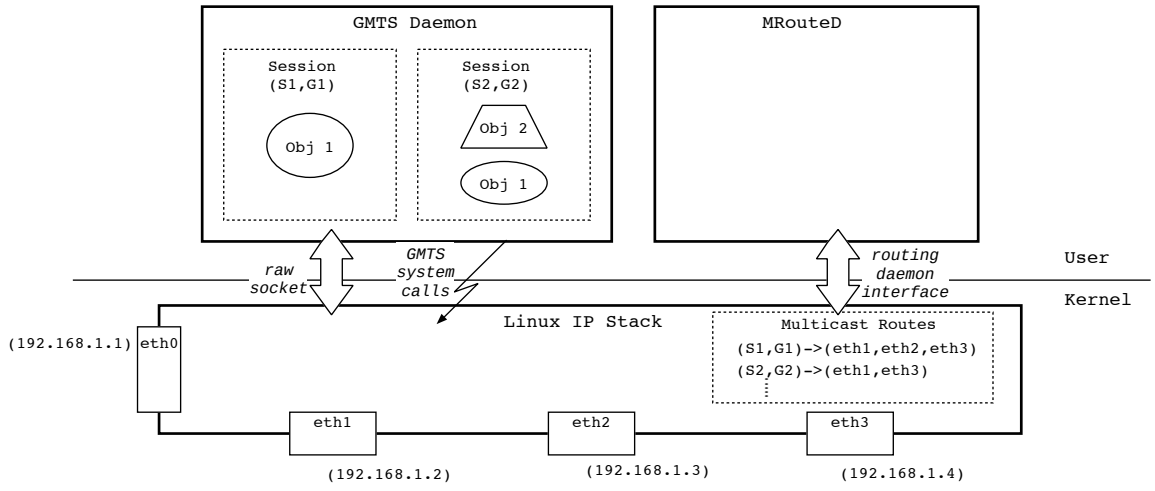

Figure 1: A block diagram of the GMTS prototype architecture showing the three main components along with the interfaces between them. We developed the GMTS daemon specifically for this prototype and made several modifications to the Linux IP stack. An *unmodified* `mrouted` multicast routing daemon is included on this diagram for completeness.

At a coarse level, the GMTS prototype is composed of three components—a GMTS

daemon operating in application space, a modified Linux kernel, and an unmodified multicast routing daemon. This basic architecture is shown in Fig. 1 The GMTS daemon processes GMTS packets and maintains the additional router state required by the GMTS protocol. In an unmodified Linux kernel configured as a router, packets are forwarded entirely within the kernel. Since the GMTS daemon must intercept and modify packets before forwarding them, our prototype uses a kernel with modified forwarding code. This modified code is invoked only for GMTS packets. The multicast routing daemon is required to process the IGMP messages and maintain standard IP multicast router state, namely a list of downstream interfaces for each combination of source and group addresses. We use `mrouted` for this purpose.[2] In the rest of this section we will provide a more detailed look at both the modified kernel and the design of the GMTS daemon.

3.1 Kernel Support

Our goal has been to implement as much of the GMTS functionality as possible in a daemon so as to minimize the number of modifications to the Linux kernel. With this in mind, we found that some kernel modifications were required to support three essential features of GMTS:

1. Deferred forwarding, pending processing in the daemon.

2. Modifying GMTS headers (and therefore checksums) of forwarded packets.

3. Subcast.

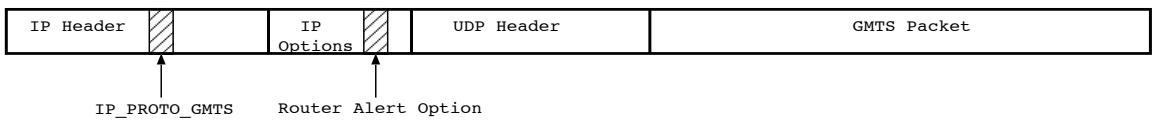

Figure 2: One possible way of implementing the GMTS within the IP protocol stack is to tag GMTS packets with the IP router alert option and introduce a special GMTS protocol identifier.

[2]Useful information about Linux implementations of `mrouted` and other multicast tools can be found at http://www.cs.washington.edu/homes/esler/multicast.

The interface between the GMTS daemon and the kernel consists of a raw socket, an extended `sockaddr` data structure, and three new system calls. GMTS packets are encapsulated in UDP packets, as shown in Fig. 2. Each GMTS packet contains a special protocol identifier in the IP header protocol field and has the IP router alert option set.[3] GMTS packets are placed in the read queue of the daemon's raw socket during standard router alert option processing in the kernel. In an unmodified Linux kernel, the inspection of router alert packets would not disrupt the forwarding of the packet; a copy of the packet would simply be placed in the queue of any interested raw socket. In the case of GMTS, however, the daemon must make a forwarding decision and overwrite a field in the GMTS header before the packet can be forwarded. We have therefore modified the Linux kernel to suspend the forwarding of GMTS packets while they are processed by the GMTS daemon. The daemon can resume forwarding in the kernel by executing one of three system calls, `gmts_fwd()`, `gmts_drop()`, and `gmts_subcast()`.

The GMTS daemon uses the `rcvfrom()` system call to obtain a copy of each GMTS protocol packet traversing the router along with a completed sockaddr_gmts structure.

```
struct if_selector {
  __u32 if_addr;
  unsigned char sel;
};

struct sockaddr_gmts {
  struct sockadd_in sin;
  void *fwd_cookie;
  int nifs;
  /* Followed in memory by an array
     of if_selectors...
     struct if_selector ifs[nifs];
  */
};
```

The `sockaddr_gmts` structure contains not only the IP address of the packets source, but also a handle to the kernel's copy of the packet stored in the `fwd_cookie` data member. The daemon must pass this handle back to the kernel in one of the three GMTS system calls.

[3]The IP router alert option is defined in RFC 2113.

These system calls enable the daemon to control the forwarding behavior by instructing the kernel to drop the packet, resume normal forwarding, or subcast the packet. The daemon must execute exactly one of these system calls for each GMTS packet processed.

The `sockaddr_gmts` structure also contains an array of `if_selector` structures, one for each virtual interface recognized by the router. The kernel uses this array to inform the daemon which interface the packet arrived on by setting that interfaces `sel` field to 1 for the arrival interface and 0 for all others. This same array is used later by the daemon to provide the kernel with a subset of interfaces on which to subcast a packet when forwarding is resumed with `gmts_subcast()`. During subcast, the kernel forwards a packet down any interface that is part of the multicast tree and has been selected by the daemon.

To illustrate a typical interaction between the kernel and daemon, consider the operation of the simple reliable multicast protocol presented in Section 2.3 implemented using the GPO. The source multicasts an SSP to initialize the GPO state in the network. GMTS packets are identified solely on the basis of the protocol identifier in the IP header and the presence of the IP router alert option. When this packet traverses a GMTS router, it is held in the kernel and a copy is passed to the routing daemon via the `rcvfrom()` system call. The daemon processes the packet, creating the GPO object in memory, and instructs the kernel to resume normal multicast forwarding using the `gmts_fwd()` system call. The kernel retrieves its copy of the packet using the pointer passed back to it by the daemon and reintroduces this packet into the kernel forwarding path. As it forwards the packet on each downstream interface, the modified kernel replaces the NEXT_HOP_ADDRESS field in the GMTS header with the IP address of the outgoing interface. Data packets multicast by the source also traverse the router, but these packets do not have the router alert option set and are therefore forwarded normally within the kernel.

The router receives repair requests in the form of MIPs unicast from downstream nodes—either from receivers with no closer GMTS router on their forwarding path or from downstream GMTS routers. These packets are also passed to the daemon for processing. The daemon uses the `if_selector` array returned from the kernel to construct the arrival vector required by the *rcvr_update* method. Since the router itself is the final destination of the MIP containing the repair request, the daemon instructs the kernel to drop the packet with

the `gmts_drop()` system call, freeing the associated memory in the kernel.[4] However, the router will unicast a similar repair request upstream to its nearest GMTS neighbor if no previous repair request has been sent for the specified sequence number.

Repair packets take the form of MIPs sent to the group multicast address by the source. As before, these packets are trapped by the daemon and processed. In this case, the router will subcast the packet only on those downstream interfaces on which a repair request previously arrived. To perform this subcast, the daemon encodes the current interface state vector as an Ω_selector array appended to a `sockaddr_gmts` structure. It passes this structure to the kernel via a call to `gmts_subcast()`. The kernel reintroduces its copy of the packet into the multicast forwarding path, but restricts forwarding to only the interfaces indicated.

The required modifications to the Linux kernel required changing only five source module and three header files, and the introduction of two additional header files. Table 2 summarizes the changes that we made to the Linux kernel source code.

3.2 GMTS Daemon

The GMTS daemon waits in its main loop for GMTS packets to be passed from the kernel via a raw socket, processing each packet as it arrives. The daemon is responsible for implementing the GMTS protocol by processing SSP and MIP packets. SSPs contain one or more GMTS objects, each uniquely identified by a combination of source address, group address, and object identifier. The daemon processes SSPs by creating an in-memory representation of each object described in the packet or refreshing the state of any objects that already exist.

The daemon is implemented in C, although its has an underlying object-oriented design. This design separates the functionality of specific GMTS object types from the general type-independent protocol processing, making it easy to introduce new object types as GMTS

[4]Although repair requests are unicast directly to the router, the kernel delivers these packets to the daemon as a result of processing the router alert IP option. This router alert processing occurs before the kernel recognizes that the packet is addressed to the local host. Since kernel processing for the packet is suspended until daemon processing completes, the daemon simply instructs the kernel to drop the packet rather than allowing it to be presented a second time.

Module	Description
net/socket.c	Implemented three new entry points for the SYS_socketcall system call for gmts_fwd(), gmts_subcast(), and gmts_drop().
net/ipv4/ip_forward.c	Added ip_reforward method to resume deferred forwarding.
net/ipv4/ip_input.c	Modified ip_call_ra_chain to support deferred forwarding.
net/ipv4/ipmr.c	New functions gmts_queue_xmit() and gmts_forward() implement subcast and modify NEXT_HOP_ADDRESS. New function ip_mr_reinput() allows deferred multicast packet to be reintroduced into forwarding path.
net/ipv4/raw.c	Actual implementation of system calls in raw_gmts_fwd(), raw_gmts_subcast(), and raw_gmts_drop(). Modified rcvmsg implementation for raw sockets and introduced new RA_NO_FORWARD socket option add support for deferred forwarding.
include/linux/gmts.h	Defines kernel-daemon interface.
include/linux/gmtspcol.h	Protocol packet data structures and constants.
include/linux/in.h	Defined IPPROTO_GMTS.
include/linux/net.h	Defined new socket system call identifiers.
include/linux/skbuff.h	Added timestamp to skbuff structure for performance measurement
include/net/ip.h	Declarations of new forwarding functions.
include/net/raw.h	Declarations of functions implementing GMTS system calls.
include/net/sock.h	New raw socket option for deferred forwarding

Table 2: Summary of modifications to the Linux kernel source code.

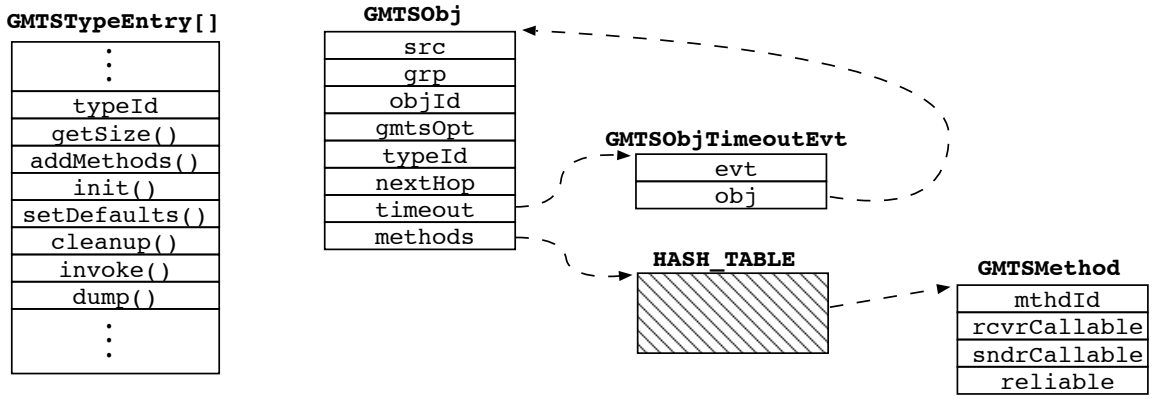

Figure 3: Data structures comprising the type-indpendent representation of GMTS objects. An array of `GMTSTypeEntry` structures indexed by `typeID` holds pointers to type-dependent functions. A `GMTSObjTimeoutEvt` structure handles object-level timeouts. Method *options* are type independent, although each object type implements different methods. Type-dependent code is responsible for adding the appropriate `GMTSMethod` structures to the hash table.

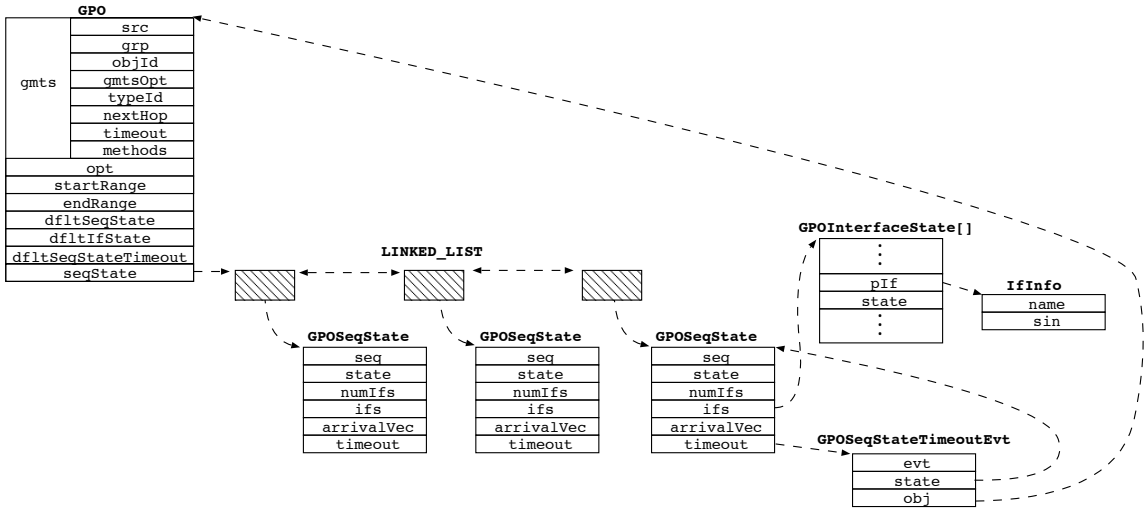

Figure 4: Data structures used to represent the GPO. The GPO maintains a linked list of sequence number data for up to a window's worth of sequence numbers. Each `GPOSeqState` structure has an individual timeout, managed by a `GPOSeqStateTimeoutEvt` structure.

evolves. In object oriented terms, all GMTS object types are derived from a virtual base class. The daemon interacts with specific object types through a set of type-specific functions implemented differently for each type. Appendix B.1 lists the `typeEntry` data structure and function definitions used to implement the type-specific function table. A global array of `typeEntry` structures contains pointers to sets of functions indexed by object type.

The in-memory representation of a GMTS object is shown graphically in Figures 3 and 4. This representation is designed to allow tasks common to all object types, such as refreshing object timeouts, to be handled by a generic object management framework. The representation for each type of object is a data structure that must contain a `GMTSObj` structure as its first member. The type-independent code only operates on this portion of the object. The type-specific functions defined in the function table operates on the remaining structure elements. The listing in Appendix B.2 shows how the GPO object type is declared in term of the abstract object structures.

A centralized timeout mechanism handles timeouts for all scheduled operations maintaining a linked list of event data structures. A standard ANSI C alarm signal is set for the earliest event. When the alarm fires, the timer manager calls raises a flag but defers actual event processing if a packet is currently being processed. During event processing an

execution procedure is invoked for any expired events through the `exec` function pointer of the `Event` data structure. The timeouts of all remaining events are adjusted and the alarm is reset. By treating events generically, the timer mechanism is able to process timeouts for objects of all types as well as for components of specific types of objects, such as sequence number data in the GPO that can expire independently. Appendix B.3 shows the abstract `Event` base class along with two concrete types of events.

The basic algorithm for processing an object contained in an SSP, shown Fig. 5, is one example of how the generic object management code interacts with type-specific operations. As the algorithm shows, the object creation process is divided between type-specific operations such as setting default variable values and generic protocol operations such as storing the object in a hash table for future retrieval.

Method invocation similarly divided. Each object type exposes its own set of methods, however, the implementation of method options such as reliable invocation is type-independent. Support for method options is not implemented in the current prototype. All methods are invoked unreliably and may be invoked by either sender or receiver. Thus method invocation in the prototype is delegated entirely to type-specific code.

4 Protocol Correctness

The first experiment we conducted was primarily to test the correctness of the simple reliable transport protocol presented in Section 2.3. The experimental setup consisted of our prototype router switching among three hosts, as shown in Fig. 6.

Since our hosts and router were connected via point-to-point Ethernet and we operated the router well under full utilization, packet losses were simulated by having each receiver drop packets independently with probability p. Only data packets were subject to losses in our experiment. In a large multicast tree, we would expect the presence of GMTS-capable routers in the tree to reduce the number of repair requests reaching the source as well as the number of unrequested repairs arriving at each receiver. Due to the small size of our testbed (one router, one source and two receivers) it is difficult to draw any quantitative results about the performance of this protocol, but we were able to verify its correctness.

```
GMTSPktSspObj pktObj; /* Packet representation of a GMTS object*/
GMTSObj obj; /* In-memory representation of a GTMS object*/
typeEntry *type; /* type-specific functions */

for each pktObj in SSP {
     type = GetTypeEntry(pktObj->typeId);
     obj = LookupObject(pktObj->src, pktObj->grp, pktObj->objId);
     if (obj == NULL) {
          /*Object does not exist. Create it.*/
          obj = AllocateObject(type->getSize());

          /*Set type-independent data*/
          obj->src = pktObj->src;
          obj->grp = pktObj->grp;
          obj->objId = pktObj->objId;
          obj->typeId = pktObj->typeId;
          SetObjectOptions(obj, pktObj->opt);
          obj->timeout = SetTimeout(pktObj->timeout);

          /*method options are type-independent, but the methods themselves
          are type-specific. Threrefore, type->addMethods() is used to fill
          the methods hash table*/
          obj->methods = CreateHashTable();
          type->addMethods(obj, obj->methods);
          for each method in pktObj {
               SetMethodOptions(Get(obj->methods, method->id) method->opt);
          }

          /*Set type-specific variables to default values*/
          type->setDefaults();

          /*Override defaults with values contained in SSP*/
          for each var in pktObj {
               type->setValue(obj, var->id, var->value);
          }

          /*Final type-specific initialization*/
          type->init();


          /*Hash the object for later use*/
          StoreObject(obj);
     } else {
          /*Object already exists. Refresh its state*/

          /*Reset options and timeout*/
          SetObjectOptions(obj, pktObj->opt);
          obj->timeout = SetTimeout(pktObj->timeout);

          /*Set method options*/
          for each method in pktObj {
               SetMethodOptions(Get(obj->methods, method->id) method->opt);
          }

          /*Set type-specific variables*/
          for each var in pktObj {
               type->setValue(obj, var->id, var->value);
          }
     }
}
```

Figure 5: A pseudo-C Algorithm for processing an object declaration contained in a SSP.

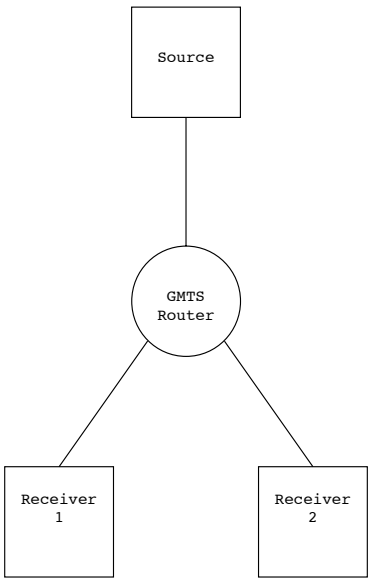

Figure 6: Configuration of testbed for verifying a simple reliable multicast protocol.

We simulated a 1 MB file transfer at a maximum packet size of 1500 bytes. A total of 732 data packets were required to complete the transfer. We deliberately slowed down the transmission of packets using a leaky bucket mechanism so that the operation of the protocol could be observed.[5] We executed this file transfer with and without the GMTS daemon in operation. In both cases, the protocol correctly delivered the entire set of sequence numbers to both receivers. However, the number of messages required for correct operation was reduced when the daemon was turned on. During each experimental run, we kept track of the following quantities: The number of NAKs received by the source n_s, the number of repair packets sent by the source r_s, the number of repair packets received by each receiver r_1 and r_2, the set of sequence numbers lost by each receiver N_1 and N_2. The quantities n_s, r_s, r_1, and r_2 are related to the sizes of the sets N_1 and N_2 differently depending on whether the

[5]The longest path in our test bed is two 100Mbps Ethernet links each under 10 feet in length. Since we operate the router at very low utilization, data packets propagate through our testbed with negligible delay. Without regulating the transmission speed it is possible for data packets to overtake the first SSP as it awaits processing at the router and reach the receivers before the GMTS object has been created. Note that it is also possible for a packet loss to occur before the SSP has been completely processed. However, a receiver will be unaware of its upstream GMTS router and the current sequence window until GMTS state has been completely established along its path to the source, such losses must go unreported.

daemon is turned on or off. Whe the daemon is turned off, all quantities are equal since the sender responds with a repair for each NAK received and no NAKs are suppressed or repairs subcast.

$$n_s = r_s = r_1 = r_2 = |N_1 \cup N_2| + |N_1 \cap N_2| \tag{5}$$

When the daemon is turned on, each one of these quantities is reduced to the minimum required for reliably communicating every sequence number to both receivers under the simplified loss model of this experiment.

$$n_s = |N_1 \cup N_2| \tag{6}$$

$$r_s = |N_1 \cup N_2| \tag{7}$$

$$r_1 = |N_1| \tag{8}$$

$$r_2 = |N_2| \tag{9}$$

We evaluated the correctness of the protocol by verifying that the above relations held in the experimental traces. When the daemon was turned off, the sender saw one NAK for each packet loss, experiencing duplicate NAKs for packets lost by both receivers. Each receiver received a repair packet for every NAK arriving at the sender, leading to unrequested repairs in cases where only one receiver lost a packet and redundant repairs when both receivers sent NAKs. When the daemon was operating, the sender received at most one NAK for each sequence number and each receiver received repairs corresponding to actual losses with no unrequested or redundant repairs. This experiment verified the GMTS router's ability to perform feedback aggregation and subcast correctly.

When performing multiple simulated file transfers back to back, we observed a bug in our implementation that may have implications for the design of the GMTS protocol. Sequence number data is typically short-lived compared to its owning object and timers associated with individual sequence number data elements are not refreshed automatically when that state is modified.[6] However, it is possible for long-lived sequence number data to interfere with protocol operation. When running two file transfers in immediate succession, for example, we found that sequence state created during the first transfer could expire during the second

[6]If need be, sequence state timers can be extended via an explicitly invoked method.

transfer between the receipt of a NAK at the router and receipt of its corresponding repair packet, causing the router to lose track of which downstream interfaces had requested the repair. In this case, the router failed to deliver the repair to any receivers and the protocol operated incorrectly. We eventually traced the source of this bug in our application to a faulty timer implementation. However, this problem can easily occur simply due to an unfortunate choice of timeout values. One way to address this problem in the protocol is to introduce a session identifier field in the GMTS header. Objects in the router would be uniquely identified by the combination of sender address, group address and session identifier, making it impossible for unexpired state from recently terminated sessions to interfere with protocol operation.

5 Performance Measurement

Although high performance was not an explicit goal of our prototype design, it is interesting to conduct a simple experiment to get a sense of the impact of GMTS processing on end-to-end performance. In a this experiment, designed to measure the overhead associated with slow-path processing of GMTS packets, we instrumented our router to record the the time required to process a GMTS packet over and above the time required for normal packet forwarding. Recall that in our prototype, each SSP and MIP must be passed from the kernel to the GMTS daemon for processing and a system call must be executed to resume forwarding within the kernel. Transferring control between the kernel and the daemon is a significant source of overhead which could certainly be reduced in a more sophisticated implementation. We are interested in measuring the total overhead introduced by our mechanism as well as how this time was divided between the kernel and the daemon.

In this experiment, we collected the appropriate timestamps for each SSP packet processed during a one MB file transfer, repeating the experiment for 30 such transfers in succession. The results are presented in Fig. 7. The figure presents processing times for successively received SSPs over all 30 transfers.

We notice that processing times are grouped into four "bands". We believe that each of these bands corresponds to a characteristic processing time depending on the extent to

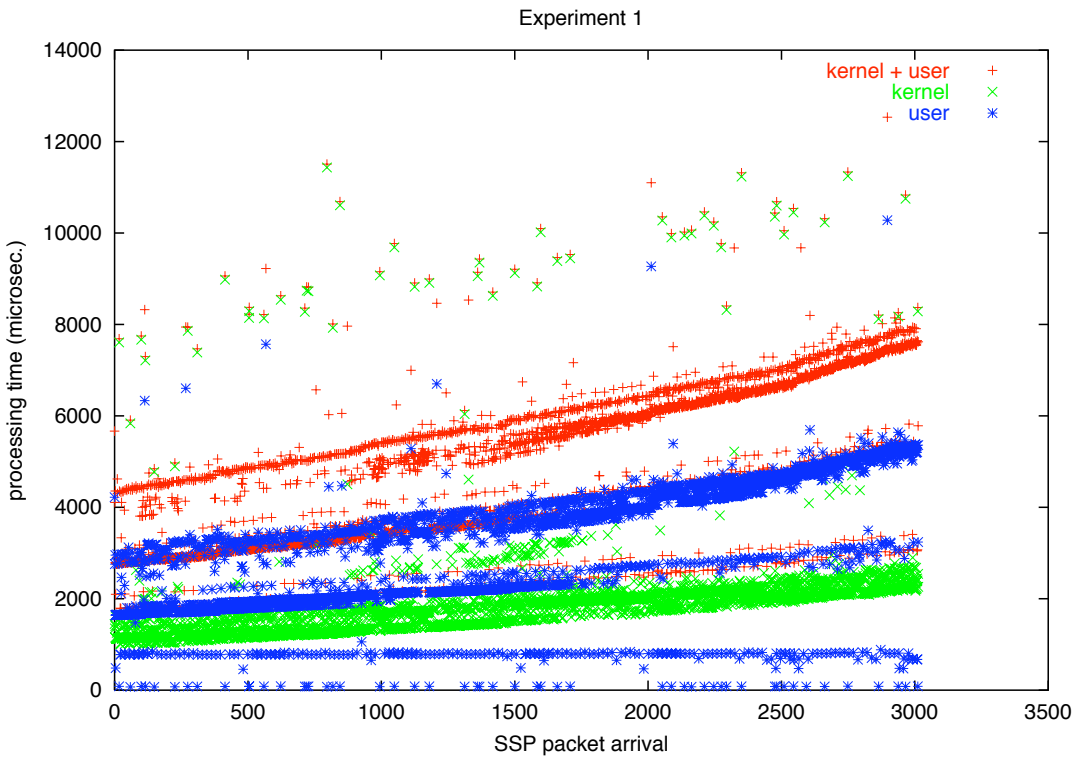

Figure 7: Total processing time for about 3000 subsequent SSP arrivals at the GMTS router. This plot shows the total processing time including the time spent transferring control from the kernel to the daemon as well as the daemon and kernel components separately. We can see the absence of the linear increase trend in the fastest daemon processing times (the horizontal band at the bottom of the figure), leading us to believe that this slowdown is introduced by the kernel.

which the daemon was interrupted by the operating system while processing an SSP. Of particular interest are the lowest two bands of the daemon data set, as these correspond to the minimum time required to perform essential GMTS slow path processing.[7]

There is a gradual linear increase in processing time that we have yet to explain. We believe this increasing overhead is introduced in the kernel itself because the fastest daemon processing times, most likely corresponding to instances of uninterrupted processing, do not exhibit this increase. When the trend is removed from the data, we observe that the time required to process a SSP in the daemon is typically about three milliseconds. An additional

[7]We conjecture that the sparse band of fastest daemon processing times represent uninterrupted processing in which much of the daemon code and GMTS object data was retrieved from cache.

one to two milliseconds is added in transferring control from the kernel to the daemon and back. In the best case, the SSP was processed in the daemon in less than one millisecond in the absence of operating system overhead. Thus sub-millisecond processing times seem to be a reasonable target for developing performance optimized code in the future.

Our performance measurements are the same order of magnitude as overheads measured by Lehman, et al. [5] in another prototype active router. We emphasize that our prototype contains no performance optimizations at all and admits numerous opportunities to reduce the processing overhead.

To understand these results in proper context, consider a typical end-to-end path in the Internet consisting of around ten hops with a total round trip time on the order of a few hundred milliseconds. Operating GMTS along such a path might increase the round trip time by between one and ten percent for (relatively infrequent) GMTS packets, depending on the number of GMTS-capable routers in the path. Ordinary data packets would not experience any additional delay.

6 Conclusion

In the process of developing this prototype, we have observed several places in which the protocol could be improved. We conclude this report by summarizing some of the lessons learned.

In is important for GMTS objects always to be constructed in a consistent state. It must therefore be possible to initialize object data and set default values using a single SSP. In the case of the GPO, for example, the object must be configured with the default timeout to assign to sequence number data. To address this need, we have extended the SSP packet format to include a variable initialization block. To enable a possible performance optimization, we have added an object sequence number to each object entry in the SSP. Incrementing this sequence number informs a GMTS router that the current SSP contains explicit changes to object variable. This allows the router to distinguish such SSPs from those that are merely intended to refresh the timer state. The revised format is shown in Appendix A.

As mentioned in Section 4, we have observed that long-lived state in the network can interfere across sessions. Appropriate setting of timeouts is important to ensure that state is removed in a timely fashion. However, introducing a session identifier in the GMTS header would enable GMTS to prevent such interference even when timeouts are set incorrectly.

Error handling in the GMTS protocol remains incompletely specified in GMTS to date. We would like to make some observations relevant to how protocol failures should be handled. It is useful in this discussion to distinguish between two failure modes, which we refer to as *internal* and *external*. External failures are due to errors in the GMTS packet, whereas internal errors are caused by unavailable resources at the router.

In the case of external errors, there is no sense in wasting network resources processing malformed packets once the error has been detected. At the same time, we would like to allow end-to-end detection of such errors. One way to do this is to introduce an error bit in a GMTS packet which is set by the first router to detect a malformed packet. Once this bit is set, the packet (if it is a multicast packet) is forwarded along the multicast tree, but not processed at any other routers. Receivers detecting such packets must notify the source. To avoid a feedback implosion due to error reporting, we propose introducing a default error object present in each GMTS router with a well-known, reserved, type and object ID. The error object would provide a method for aggregating error conditions back toward the source. We have yet to fully specify such an object. In general, having a reserved type identifier for "system" objects combined with well known object identifiers for different system objects would create a reasonably sized ID space for introducing new system objects as GMTS evolves.

Internal errors are handled differently. Consider a case in which a router has insufficient resources to create a new object in an SSP. In this case, the router should not mark the error bit so that downstream routers will still try to create the object. It is important that a router with insufficient resources to leave the NEXT_HOP_ADDRESS field of the GMTS header unchanged when forwarding the SSP. The router will simply not participate in the transport protocol managed by this object.

Our experience designing and implementing the GMTS prototype router provides support for the idea that such services can be implemented in routers easily and with only a modest

impact end-to-end protocol performance. The presence of services such as those proposed by GMTS would enable a range of scalable multicast transport protocols by making it easy to perform functions like reliable transfer, congestion control and polling.

In future work, we intend to consider the impact of a large aggregate of GMTS session on the performance of routers, as well as the impact of incomplete deployment of GMTS within the network on the effectiveness of the transport protocols it supports. We would also like to investigate how GMTS services can be implemented in high-performance hardware routers.

References

[1] B. Cain and D. Towsley. Generic multicast transport services: Router support for multicast applications. Technical report, University of Massachusetts, 1999.

[2] B. Cain and D. Towsley. Generic multicast transport services: Router support for multicast applications. In *Proc. Networking 2000*, May 2000.

[3] T. Speakman et al. Pgm reliable transport protocol. Technical report, IETF, 2000.

[4] S. Floyd, V. Jacobson, S McCanne, C. Lin, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Trans. on Networking*, 5:784–803, Dec. 1997.

[5] L. H. Lehman, S. J. Garland, and D. L. Tennenhouse. Active reliable multicast. In *Proc. INFOCOM'98*, 1998.

[6] B. N. Levine and J. J. Garcia-Luna-Aceves. Improving internet multicast with routing labels. In *Proc. ICNP-97*, pages 241–250, Oct. 1997.

[7] J. Lin and S. Paul. Rmtp: A reliable multicast transport protocol. In *Proc. INFOCOM'95*, 1995.

[8] C. Papadopoulos and G. Parulkar. An error control scheme for large-scale multicast applications. In *Proc. INFOCOM'98*, 1998.

[9] D. Towsley, J. Kurose, and S. Pingali. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. *IEEE JSAC*, April 1997.

[10] K. Yano and S. McCanne. The breadcrumb forwarding service: A synthesis of pgm and express to improve and simplify global ip multicast. *ACM Computer Communication Review*, Vol.30(No.2), 2000.

A Revised GMTS Packet Formats

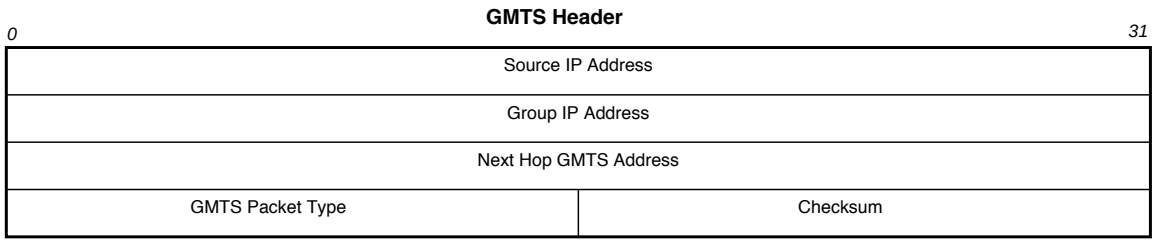

Figure 8: The GMTS packet header shared by both SSP and MIP packet types

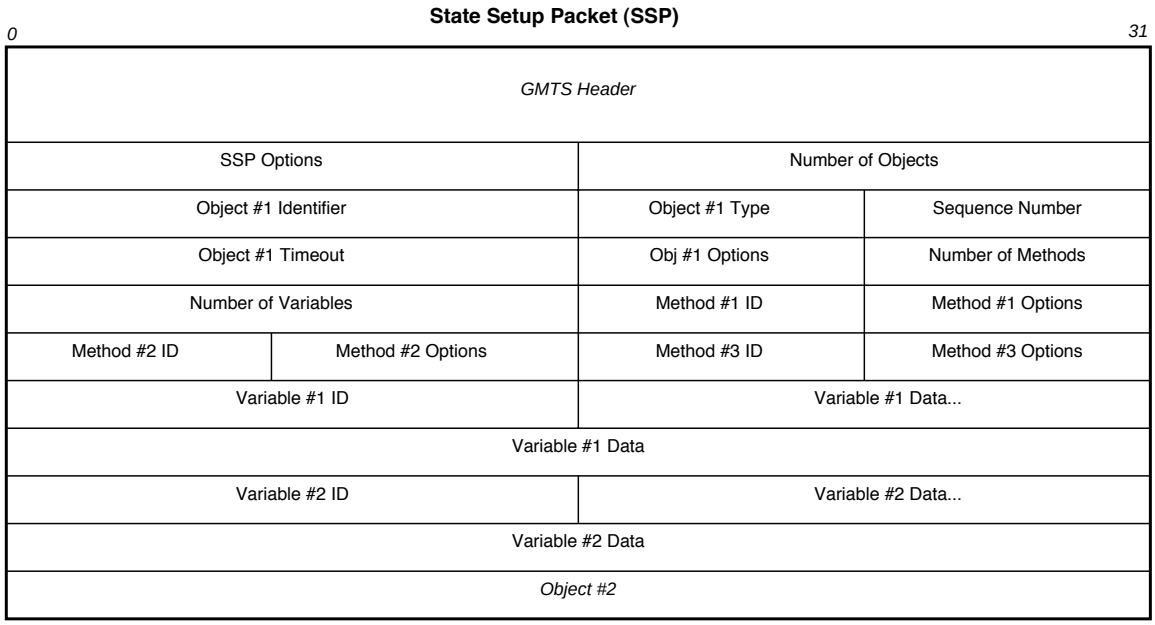

Figure 9: Revised State Setup Packet (SSP) format. The main revision is the introduction of a variable intialization block in the SSP. An object sequence number has been added to the object declaration in the SSP to tag SSPs that explicitly modify variables or options of existing objects.

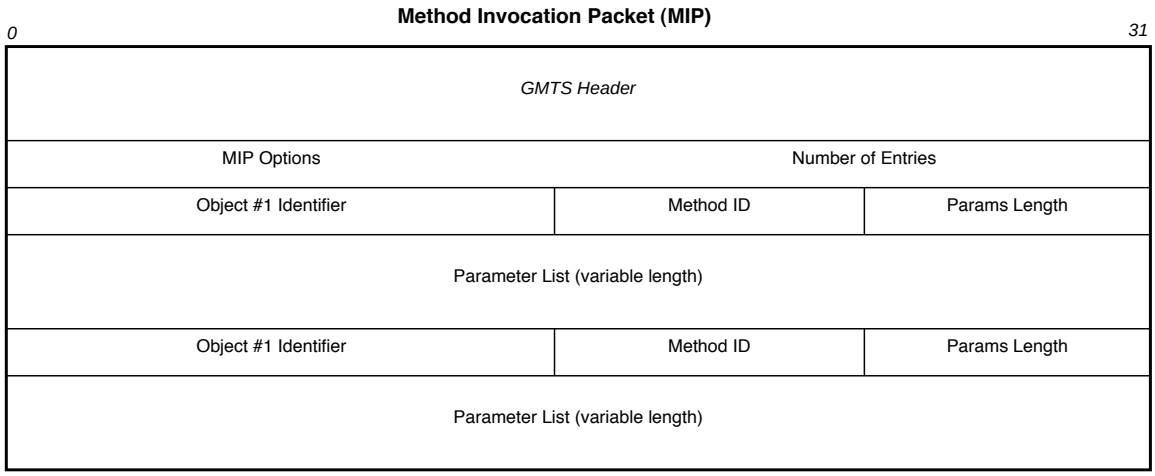

Figure 10: Method Invocation Packet (MIP) format.

B Selected GMTS Daemon Data Structures

B.1 Function Table for GMTS Object Types

```
/*GMTS Object Function Prototypes*/
typedef int (*GMTSObjAllocSizeProc) ();
typedef int (*GMTSObjAddMethodsProc) (GMTSObj *);
typedef int (*GMTSObjDfltProc) (GMTSObj *);
typedef int (*GMTSObjInitProc) (GMTSObj *, struct _GMTSPktSspObj *);
typedef void (*GMTSObjCleanupProc) (GMTSObj *);
typedef int (*GMTSObjMethodProc) (GMTSObj *, u_char,  char *, unsigned int, RequestHandl
typedef int (*GMTSObjValueProc) (GMTSObj *, u_short, char *, unsigned int);
typedef void (*GMTSObjPrintProc) (GMTSObj *);


/*'Virtual Function Table' for a GMTS object*/
typedef struct _GMTSTypeEntry {
  u_short typeId;
  GMTSObjAllocSizeProc getSize;
  GMTSObjAddMethodsProc addMethods;
  GMTSObjInitProc init;
  GMTSObjDfltProc setDefaults;
  GMTSObjCleanupProc cleanup;
  GMTSObjMethodProc invoke;
  GMTSObjValueProc setValue;
  GMTSObjPrintProc dump;
} GMTSTypeEntry;
```

B.2 In-Memory Object Representation

```
/*GMTS Method*/
typedef struct _GMTSMethod {
    u_char mthdId;
    int rcvrCallable;
    int sndrCallable;
    int reliable;
} GMTSMethod;

/*GMTS object 'base class'*/
typedef struct _GMTSObj {
    unsigned int src;
    unsigned int grp;
    unsigned short objId;
```

```
    unsigned int gmtsOpt;
    unsigned int typeId;
    unsigned int nextHop;
    EventHandle timeout;
    HASH_TABLE *methods;   /*Contains GMTSMethod structures*/
} GMTSObj;

/*GMTS general puropose object*/
typedef struct _GPO {
  GMTSObj gmts;
  unsigned int opt;
  unsigned int startRange;
  unsigned int endRange;
  unsigned int dfltSeqState;
  unsigned int dfltIfState;
  unsigned short dfltSeqStateTimeout;
  LINKED_LIST *seqState;
} GPO;

/*Sequence number state*/
typedef struct _GPOSeqState {
  unsigned int seq;
  unsigned int state;
  unsigned int numIfs;
  GPOInterfaceState *ifs;
  int *arrivalVec;       /*Temporary storage of arrival vector*/
  EventHandle timeout;
} GPOSeqState;

typedef struct _IfInfo {
  char name[IFNAMSIZ];
  struct sockaddr_in sin;
} IfInfo;

/*Interface state*/
typedef struct _GPOInterfaceState {
  struct _IfInfo *pIf;
  unsigned int state;
} GPOInterfaceState;
```

B.3 Timer-Related Data Structures

```
/*Timer Event ``base class''*/
typedef struct _Event {
```

```
    struct timeval timeout;
    char errorString[EVT_ERROR_STRING_LEN];
    EvtExecuteProc exec;
} Event;

/*Object Timeout Event*/
typedef struct _GMTSObjTimeoutEvt {
    Event evt;
    GMTSObj *obj;
} GMTSObjTimeoutEvt;


/*Sequence Number State Timeout Event*/
typedef struct _GPOSeqStateTimeoutEvt {
    Event *evt;
    GPOSeqState *state;
    GPO *obj;
} GPOSeqStateTimeoutEvt;
```