Network Working Group
Request for Comments: 1045

David Cheriton
Stanford University
February 1988

VMTP: VERSATILE MESSAGE TRANSACTION PROTOCOL
Protocol Specification

STATUS OF THIS MEMO

This RFC describes a protocol proposed as a standard for the Internet community. Comments are encouraged. Distribution of this document is unlimited.

OVERVIEW

This memo specifies the Versatile Message Transaction Protocol (VMTP) [Version 0.7 of 19-Feb-88], a transport protocol specifically designed to support the transaction model of communication, as exemplified by remote procedure call (RPC). The full function of VMTP, including support for security, real-time, asynchronous message exchanges, streaming, multicast and idempotency, provides a rich selection to the VMTP user level. Subsettability allows the VMTP module for particular clients and servers to be specialized and simplified to the services actually required. Examples of such simple clients and servers include PROM network bootload programs, network boot servers, data sensors and simple controllers, to mention but a few examples.

Table of Contents

List of Figures

1. Introduction

The Versatile Message Transaction Protocol (VMTP) is a transport protocol designed to support remote procedure call (RPC) and general transaction-oriented communication. By transaction-oriented communication, we mean that:

- Communication is request-response: A client sends a request for a service to a server, the request is processed, and the server responds. For example, a client may ask for the next page of a file as the service. The transaction is terminated by the server responding with the next page.

- A transaction is initiated as part of sending a request to a server and terminated by the server responding. There are no separate operations for setting up or terminating associations between clients and servers at the transport level.

- The server is free to discard communication state about a client between transactions without causing incorrect behavior or failures.

The term message transaction (or transaction) is used in the reminder of this document for a request-response exchange in the sense described above.

VMTP handles the error detection, retransmission, duplicate suppression and, optionally, security required for transport-level end-to-end reliability.

The protocol is designed to provide a range of behaviors within the transaction model, including:

- Minimal two packet exchanges for short, simple transactions.

- Streaming of multi-packet requests and responses for efficient data transfer.

- Datagram and multicast communication as an extension of the transaction model.

Example Uses:

- Page-level file access - VMTP is intended as the transport level for file access, allowing simple, efficient operation on a local network. In particular, VMTP is appropriate for use by diskless workstations accessing shared network file

servers.

- Distributed programming - VMTP is intended to provide an efficient transport level protocol for remote procedure call implementations, distributed object-oriented systems plus message-based systems that conform to the request-response model.

- Multicast communication with groups of servers to: locate a specific object within the group, update a replicated object, synchronize the commitment of a distributed transaction, etc.

- Distributed real-time control with prioritized message handling, including datagrams, multicast and asynchronous calls.

The protocol is designed to operate on top of a simple unreliable datagram service, such as is provided by IP.

1.1. Motivation

VMTP was designed to address three categories of deficiencies with existing transport protocols in the Internet architecture. We use TCP as the key current transport protocol for comparison.

1.1.1. Poor RPC Performance

First, current protocols provide poor performance for remote procedure call (RPC) and network file access. This is attributable to three key causes:

- TCP requires excessive packets for RPC, especially for isolated calls. In particular, connection setup and clear generates extra packets over that needed for VMTP to support RPC.

- TCP is difficult to implement, speaking purely from the empirical experience over the last 10 years. VMTP was designed concurrently with its implementation, with focus on making it easy to implement and providing sensible subsets of its functionality.

- TCP handles packet loss due to overruns poorly. We claim that overruns are the key source of packet loss in a high-performance RPC environment and, with the increasing

performance of networks, will continue to be the key source. (Older machines and network interfaces cannot keep up with new machines and network interfaces. Also, low-end network interfaces for high-speed networks have limited receive buffering.)

VMTP is designed for ease of implementation and efficient RPC. In addition, it provides selective retransmission with rate-based flow control, thus addressing all of the above issues.

1.1.2. Weak Naming

Second, current protocols provide inadequate naming of transport-level endpoints because the names are based on IP addresses. For example, a TCP endpoint is named by an Internet address and port identifier. Unfortunately, this makes the endpoint tied to a particular host interface, not specifically the process-level state associated with the transport-level endpoint. In particular, this form of naming causes problems for process migration, mobile hosts and multi-homed hosts. VMTP provides host-address independent names, thereby solving the above mentioned problems.

In addition, TCP provides no security and reliability guarantees on the dynamically allocated names. In particular, other than well-known ports, (host-addr, port-id)-tuples can change meaning on reboot following a crash. VMTP provides large identifiers with guarantee of stability, meaning that either the identifier never changes in meaning or else remains invalid for a significant time before becoming valid again.

1.1.3. Function Poor

TCP does not support multicast, real-time datagrams or security. In fact, it only supports pair-wise, long-term, streamed reliable interchanges. Yet, multicast is of growing importance and is being developed for the Internet (see RFC 966 and 988). Also, a datagram facility with the same naming, transmission and reception facilities as the normal transport level is a powerful asset for real-time and parallel applications. Finally, security is a basic requirement in an increasing number of environments. We note that security is natural to implement at the transport level to provide end-to-end security (as opposed to (inter)network level security). Without security at the transport level, a transport level protocol cannot guarantee the standard transport level service definition in the presence of an intruder. In particular, the intruder can interject packets or modify

packets while updating the checksum, making mockery out of the transport-level claim of "reliable delivery".

In contrast, VMTP provides multicast, real-time datagrams and security, addressing precisely these weaknesses.

In general, VMTP is designed with the next generation of communication systems in mind. These communication systems are characterized as follows. RPC, page-level file access and other request-response behavior dominates. In addition, the communication substrate, both local and wide-area, provides high data rates, low error rates and relatively low delay. Finally, intelligent, high-performance network interfaces are common and in fact required to achieve performance that approximates the network capability. However, VMTP is also designed to function acceptably with existing networks and network interfaces.

1.2. Relation to Other Protocols

VMTP is a transport protocol that fits into the layered Internet protocol environment. Figure 1-1 illustrates the place of VMTP in the protocol hierarchy.

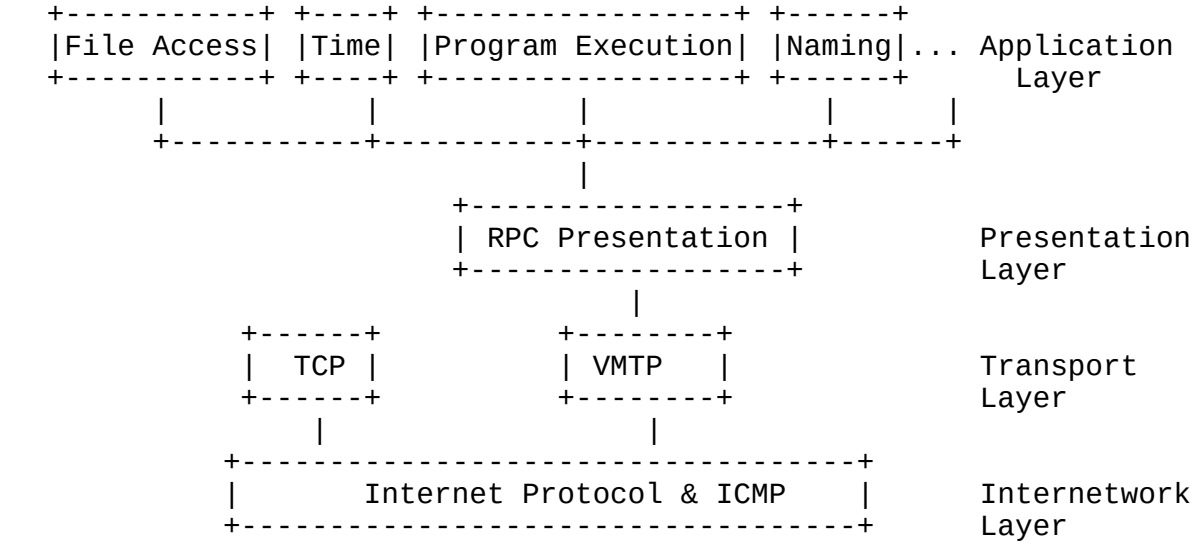


Figure 1-1: Relation to Other Protocols

The RPC presentation level is not currently defined in the Internet suite of protocols. Appendix II defines a proposed RPC presentation level for use with VMTP and assumed for the definition of the VMTP management procedures. There is also a need for the definition of the

Application layer protocols listed above.

If internetwork services are not required, VMTP can be used without the IP layer, layered directly on top of the network or data link layers.

1.3. Document Overview

The next chapter gives an overview of the protocol, covering naming, message structure, reliability, flow control, streaming, real-time, security, byte-ordering and management. Chapter 3 describes the VMTP packet formats. Chapter 4 describes the client VMTP protocol operation in terms of pseudo-code for event handling. Chapter 5 describes the server VMTP protocol operation in terms of pseudo-code for event handling. Chapter 6 summarizes the state of the protocol, some remaining issues and expected directions for the future. Appendix I lists some standard Response codes. Appendix II describes the RPC presentation protocol proposed for VMTP and used with the VMTP management procedures. Appendix III lists the VMTP management procedures. Appendix IV proposes initial approaches for handling entity identification for VMTP. Appendix V proposes initial authentication domains for VMTP. Appendix VI provides some details for implementing VMTP on top of IP. Appendix VII provides some suggestions on host implementation of VMTP, focusing on data structures and support functions. Appendix VIII describes a proposed program interface for UNIX 4.3 BSD and its descendants and related systems.

2. Protocol Overview

VMTP provides an efficient, reliable, optionally secure transport service in the message transaction or request-response model with the following features:

- Host address-independent naming with provision for multiple forms of names for endpoints as well as associated (security) principals. (See Sections 2.1, 2.2, 3.1 and Appendix IV.)

- Multi-packet request and response messages, with a maximum size of 4 megaoctets per message. (Sections 2.3 and 2.14.)

- Selective retransmission. (Section 2.13.) and rate-based flow control to reduce overrun and the cost of overruns. (Section 2.5.6.)

- Secure message transactions with provision for a variety of encryption schemes. (Section 2.6.)

- Multicast message transactions with multiple response messages per request message. (Section 2.7.)

- Support for real-time communication with idempotent message transactions with minimal server overhead and state (Section 2.5.3), datagram request message transactions with no response, optional header-only checksum, priority processing of transactions, conditional delivery and preemptive handling of requests (Section 2.8)

- Forwarded message transactions as an optimization for certain forms of nested remote procedure calls or message transactions. (Section 2.9.)

- Multiple outstanding (asynchronous) message transactions per client. (Section 2.11.)

- An integrated management module, defined with a remote procedure call interface on top of VMTP providing a variety of communication services (Section 2.10.)

- Simple subset implementation for simple clients and simple servers. (Section 2.16.)

This chapter provides an overview of the protocol as introduction to the basic ideas and as preparation for the subsequent chapters that describe the packet formats and event processing procedures in detail.

In overview, VMTP provides transport communication between network-visible entities via message transactions. A message transaction consists of a request message sent by the client, or requestor, to a group of server entities followed by zero or more response messages to the client, at most one from each server entity. A message is structured as a message control portion and a segment data portion. A message is transmitted as one or more packet groups. A packet group is one or more packets (up to a maximum of 32 packets) grouped by the protocol for acknowledgment, sequencing, selective retransmission and rate control.

Entities and VMTP operations are managed using a VMTP management mechanism that is accessed through a procedural interface (RPC) implemented on top of VMTP. In particular, information about a remote entity is obtained and maintained using the Probe VMTP management operation. Also, acknowledgment information and requests for retransmission are sent as notify requests to the management module. (In the following description, reference to an "acknowledgment" of a request or a response refers to a management-level notify operation that is acknowledging the request or response.)

2.1. Entities, Processes and Principals

VMTP defines and uses three main types of identifiers: entity identifiers, process identifiers and principal identifiers, each 64-bits in length. Communication takes place between network-visible entities, typically mapping to, or representing, a message port or procedure invocation. Thus, entities are the VMTP communication endpoints. The process associated with each entity designates the agent behind the communication activity for purposes of resource allocation and management. For example, when a lock is requested on a file, the lock is associated with the process, not the requesting entity, allowing a process to use multiple entity identifiers to perform operations without lock conflict between these entities. The principal associated with an entity specifies the permissions, security and accounting designation associated with the entity. The process and principal identifiers are included in VMTP solely to make these values available to VMTP users with the security and efficiency provided by VMTP. Only the entity identifiers are actively used by the protocol.

Entity identifiers are required to have three properties;

Uniqueness — Each entity identifier is uniquely defined at any given time. (An entity identifier may be reused over time.)

Stability — An entity identifier does not change between valid

meanings without suitable provision for removing references to the entity identifier. Certain entity identifiers are strictly stable, (i.e. never changing meaning), typically being administratively assigned (although they need not be bound to a valid entity at all times), often called well-known identifiers. All other entity identifiers are required to be T-stable, not change meaning without having remained invalid for at least a time interval T.

Host address independent
An entity identifier is unique independent of the host address of its current host. Moreover, an entity identifier is not tied to a single Internet host address. An entity can migrate between hosts, reside on a mobile host that changes Internet addresses or reside on a multi-homed host. It is up to the VMTP implementation to determine and maintain up to date the host addresses of entities with which it is communicating.

The stability of entity identifiers guarantees that an entity identifier represents the same logical communication entity and principal (in the security sense) over the time that it is valid. For example, if an entity identifier is authenticated as having the privileges of a given user account, it continues to have those privileges as long as it is continuously valid (unless some explicit notice is provided otherwise). Thus, a file server need not fully authenticate the entity on every file access request. With T-stable identifiers, periodically checking the validity of an entity identifier with period less than T seconds detects a change in entity identifier validity.

A group of entities can form an entity group, which is a set of zero or more entities identified by a single entity identifier. For example, one can have a single entity identifier that identifies the group of name servers. An entity identifier representing an entity group is drawn from the same name space as entity identifiers. However, single entity identifiers are flagged as such by a bit in the entity identifier, indicating that the identifier is known to identify at most one entity. In addition to the group bit, each entity identifier includes other standard type flags. One flag indicates whether the identifier is an alias for an entity in another domain (See Section 2.2 below.). Another flag indicates, for an entity group identifier, whether the identifier is a restricted group or not. A restricted group is one in which an entity can be added only by another entity with group management authorization. With an unrestricted group, an entity is allowed to add itself. If an entity identifier does not represent a

group, a type bit indicates whether the entity uses big-endian or little-endian data representation (corresponding to Motorola 680X0 and VAX byte orders, respectively). Further specification of the format of entity identifiers is contained in Section 3.1 and Appendix IV.

An entity identifier identifies a Client, a Server or a group of Servers <1>. A Client is always identified by a T-stable identifier. A server or group of servers may be identified by a a T-stable identifier (group or single entity) or by strictly stable (statically assigned) entity group identifier. The same T-stable identifier can be used to identify a Client and Server simultaneously as long as both are logically associated with the same entity. The state required for reliable, secure communication between entities is maintained in client state records (CSRs), which include the entity identifier of the Client, its principal, its current or next transaction identifier and so on.

2.2. Entity Domains

An entity domain is an administration or an administration mechanism that guarantees the three required entity identifier properties of uniqueness, stability and host address independence for the entities it administers. That is, entity identifiers are only guaranteed to be unique and stable within one entity domain. For example, the set of all Internet hosts may function as one domain. Independently, the set of hosts local to one autonomous network may function as a separate domain. Each entity domain is identified by an entity domain identifier, Domain. Only entities within the same domain may communicate directly via VMTP. However, hosts and entities may participate in multiple entity domains simultaneously, possibly with different entity identifiers. For example, a file server may participate in multiple entity domains in order to provide file service to each domain. Each entity domain specifies the algorithms for allocation, interpretation and mapping of entity identifiers.

Domains are necessary because it does not appear feasible to specify one universal VMTP entity identification administration that covers all entities for all time. Domains limit the number of entities that need to be managed to maintain the uniqueness and stability of the entity

---

<1> Terms such as Client, Server, Request, Response, etc. are capitalized in this document when they refer to their specific meaning in VMTP.

name space. Domains can also serve to separate entities of different security levels. For instance, allocation of a unclassified entity identifier cannot conflict with secret level entity identifiers because the former is interpreted only in the unclassified domain, which is disjoint from the secret domain.

It is intended that there be a small number of domains. In particular, there should be one (or a few) domains per installation "type", rather than per installation. For example, the Internet is expected to use one domain per security level, resulting in at most 8 different domains. Cluster-based internetwork architectures, those with a local cluster protocol distinct from the wide-area protocol, may use one domain for local use and one for wide-area use.

Additional details on the specification of specific domains is provided in Appendix IV.

2.3. Message Transactions

The message transaction is the unit of interaction between a Client that initiates the transaction and one or more Servers. A message transaction starts with a request message generated by a client. At the service interface, a server becomes involved with a transaction by receiving and accepting the request. A server terminates its involvement with a transaction by sending a response message. In a group message transaction, the server entity designated by the client corresponds to a group of entities. In this case, each server in the group receives a copy of the request. In the client’s view, the transaction is terminated when it receives the response message or, in the case of a group message transaction, when it receives the last response message. Because it is normally impractical to determine when the last response message has been received. the current transaction is terminated by VMTP when the next transaction is initiated.

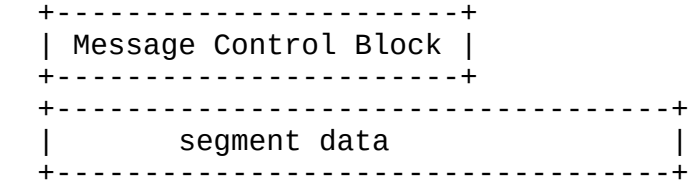
Within an entity domain, a transaction is uniquely identified by the tuple (Client, Transaction, ForwardCount). where Transaction is a 32-bit number and ForwardCount is a 4-bit value. A Client uses monotonically increasing Transaction identifiers for new message transactions. Normally, the next higher transaction number, modulo 2**32, is used for the next message transaction, although there are cases in which it skips a small range of Transaction identifiers. (See the description of the STI control flag.) The ForwardCount is used when a message transaction is forwarded and is zero otherwise.

A Client generates a stream of message transactions with increasing transaction identifiers, directed at a diversity of Servers. We say a

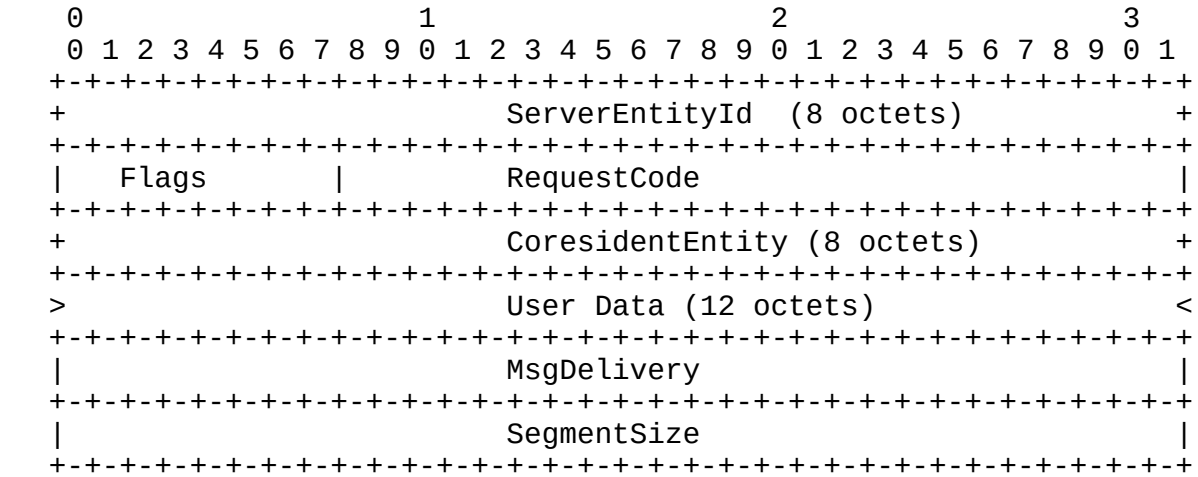
Client has a transaction outstanding if it has invoked a message transaction, but has not received the last Response (or possibly any Response). Normally, a Client has only one transaction outstanding at a time. However, VMTP allows a Client to have multiple message transactions outstanding simultaneously, supporting streamed, asynchronous remote procedure call invocations. In addition, VMTP supports nested calls where, for example, procedure A calls procedure B which calls procedure C, each on a separate host with different client entity identifiers for each call but identified with the same process and principal.

2.4. Request and Response Messages

A message transaction consists of a request message and one or more Response messages. A message is structured as message control block (MCB) and segment data, passed as parameters, as suggested below.



In the request message, the MCB specifies control information about the request plus an optional data segment. The MCB has the following format:



The ServerEntityId is the entity to which the Request MCB is to be sent (or was sent, in the case of reception). The Flags indicate various options in the request and response handling as well as whether the

CoresidentEntity, MsgDelivery and SegmentSize fields are in use. The RequestCode field specifies the type of Request. It is analogous to a packet type field of the Ethernet, acting as a switch for higher-level protocols. The CoresidentEntity field, if used, designates a subgroup of the ServerEntityId group to which the Request should be routed, namely those members that are co-resident with the specified entity (or entity group). The primary intended use is to specify the manager for a particular service that is co-resident with a particular entity, using the well-known entity group identifier for the service manager in the ServerEntityId field and the identifier for the entity in the CoresidentEntity field. The next 12 octets are user- or application-specified.

The MsgDelivery field is optionally used by the RPC or user level to specify the portions of the segment data to transmit and on reception, the portions received. It provides the client and server with (optional) access to, and responsibility for, a simple selective transmission and reception facility. For example, a client may request retransmission of just those portions of the segment that it failed to receive as part of the original Response. The primary intended use is to support highly efficient multi-packet reading from a file server. Exploiting user-level selective retransmission using the MsgDelivery field, the file server VMTP module need not save multi-packet Responses for retransmission. Retransmissions, when needed, are instead handled directly from the file server buffers.

The SegmentSize field indicates the size of the data segment, if present. The CoresidentEntity, MsgDelivery and SegmentSize fields are usable as additional user data if they are not otherwise used.

The Flags field provides a simple mechanism for the user level to communicate its use of VMTP options with the VMTP module as well as for VMTP modules to communicate this use among themselves. The use of these options is generally fixed for each remote procedure so that an RPC mechanism using VMTP can treat the Flags as an integral part of the RequestCode field for the purpose of demultiplexing to the correct stub.

A Response message control block follows the same format except the Response is sent from the Server to the Client and there is no Coresident Entity field (and thus 20 octets of user data).

2.5. Reliability

VMTP provides reliable, sequenced transfer of request and response messages as well as several variants, such as unreliable datagram requests. The reliability mechanisms include: transaction identifiers,

checksums, positive acknowledgment of messages and timeout and retransmission of lost packets.

2.5.1. Transaction Identifiers

Each message transaction is uniquely identified by the pair (Client, Transaction). (We defer discussion of the ForwardCount field to Section 2.9.) The 32-bit transaction identifier is initialized to a random value when the Client entity is created or allocated its entity identifier. The transaction identifier is incremented at the end of each message transaction. All Responses with the same specified (Client, Transaction) pair are associated with this Request.

The transaction identifier is used for duplicate suppression at the Server. A Server maintains a state record for each Client for which it is processing a Request, identified by (Client, Transaction). A Request with the same (Client, Transaction) pair is discarded as a duplicate. (The ForwardCount field must also be equal.) Normally, this record is retained for some period after the Response is sent, allowing the Server to filter out subsequent duplicates of this Request. When a Request arrives and the Server does not have a state record for the sending Client, the Server takes one of three actions:

1. The Server may send a Probe request, a simple query operation, to the VMTP management module associated with the requesting Client to determine the Client's current Transaction identifier (and other information), initialize a new state record from this information, and then process the Request as above.

2. The Server may reason that the Request must be a new request because it does not have a state record for this Client if it keeps these state records for the maximum packet lifetime of packets in the network (plus the maximum VMTP retransmission time) and it has not been rebooted within this time period. That is, if the Request is not new either the Request would have exceeded the maximum packet lifetime or else the Server would have a state record for the Client.

3. The Server may know that the Request is idempotent or can be safely redone so it need not care whether the Request is a duplicate or not. For example, a request for the current time can be responded to with the current time without being concerned whether the Request is a duplicate. The Response is discarded at the Client if it is no longer of interest.

2.5.2. Checksum

Each VMTP packet contains a checksum to allow the receiver to detect corrupted packets independent of lower level checks. The checksum field is 32 bits, providing greater protection than the standard 16-bit IP checksum (in combination with an improved checksum algorithm). The large packets, high packet rates and general network characteristics expected in the future warrant a stronger checksum mechanism.

The checksum normally covers both the VMTP header and the segment data. Optionally (for real-time applications), the checksum may apply only to the packet header, as indicated by the HCO control bit being set in the header. The checksum field is placed at the end of the packet to allow it to be calculated as part of a software copy or as part of a hardware transmission or reception packet processing pipeline, as expected in the next generation of network interfaces. Note that the number of header and data octets is an integral multiple of 8 because VMTP requires that the segment data be padded to be a multiple of 64 bits. The checksum field is appended after the padding, if any. The actual algorithm is described in Section 3.2.

A zero checksum field indicates that no checksum was transmitted with the packet. VMTP may be used without a checksum only when there is a host-to-host error detection mechanism and the VMTP security facility is not being used. For example, one could rely on the Ethernet CRC if communication is restricted to hosts on the same Ethernet and the network interfaces are considered sufficiently reliable.

2.5.3. Request and Response Acknowledgment

VMTP assumes an unreliable datagram network and internetwork interface. To guarantee delivery of Requests and Response, VMTP uses positive acknowledgments, retransmissions and timeouts.

A Request is normally acknowledged by receipt of a Response associated with the Request, i.e. with the same (Client, Transaction). With streamed message transactions, it may also be acknowledged by a subsequent Response that acknowledges previous Requests in addition to the transaction it explicitly identifies. A Response may be explicitly acknowledged by a NotifyVmtpServer operation requested of the manager for the Server. In the case of streaming, this is a cumulative acknowledgment, acknowledging all Responses with a lower transaction identifier as well.) In addition, with non-streamed communication, a subsequent Request from the same Client acknowledges Responses to all previous message transactions (at least in the sense that either the client received a Response or is no longer interested in Responses to

those earlier message transactions). Finally, a client response timeout (at the server) acknowledges a Response at least in the sense that the server need not be prepared to retransmit the Response subsequently. Note that there is no end-to-end guarantee of the Response being received by the client at the application level.

2.5.4. Retransmissions

In general, a Request or Response is retransmitted periodically until acknowledged as above, up to some maximum number of retransmissions. VMTP uses parameters RequestRetries(Server) and ResponseRetries(Client) that indicate the number of retransmissions for the server and client respectively before giving up. We suggest the value 5 be used for both parameters based on our experience with VMTP and Internet packet loss. Smaller values (such as 3) could be used in low loss environments in which fast detection of failed hosts or communication channels is required. Larger values should be used in high loss environments where transport-level persistence is important.

In a low loss environment, a retransmission only includes the MCB and not the segment data of the Request or Response, resulting in a single (short) packet on retransmission. The intended recipient of the retransmission can request selective retransmission of all or part of the segment data as necessary. The selective retransmission mechanism is described in Section 2.13.

If a Response is specified as idempotent, the Response is neither retransmitted nor stored for retransmission. Instead, the Client must retransmit the Request to effectively get the Response retransmitted. The server VMTP module responds to retransmissions of the Request by passing the Request on to the server again to have it regenerate the Response (by redoing the operation), rather than saving a copy of the Response. Only Request packets for the last transaction from this client are passed on in this fashion; older Request packets from this client are discarded as delayed duplicates. If a Response is not idempotent, the VMTP module must ensure it has a copy of the Response for retransmission either by making a copy of the Response (either physically or copy-on-write) or by preventing the Server from continuing until the Response is acknowledged.

2.5.5. Timeouts

There is one client timer for each Client with an outstanding transaction. Similarly, there is one server timer for each Client transaction that is "active" at the server, i.e. there is a transaction

record for a Request from the Client.

When the client transmits a new Request (without streaming), the client timer is set to roughly the time expected for the Response to be returned. On timeout, the Request is retransmitted with the APG (Acknowledge Packet Group) bit set. The timeout is reset to the expected roundtrip time to the Server because an acknowledgment should be returned immediately unless a Response has been sent. The Request may also be retransmitted in response to receipt of a VMTP management operation indicating that selected portions of the Request message segment need to be retransmitted. With streaming, the timeout applies to the oldest outstanding message transaction in the run of outstanding message transactions. Without streaming, there is one message transaction in the run, reducing to the previous situation. After the first packet of a Response is received, the Client resets the timeout to be the time expected before the next packet in the Response packet group is received, assuming it is a multi-packet Response. If not, the timer is stopped. Finally, the client timer is used to timeout waiting for second and subsequent Responses to a multicast Request.

The client timer is set at different times to four different values:

TC1(Server) The expected time required to receive a Response from the Server. Set on initial Request transmission plus after its management module receives a NotifyVmtpClient operation, acknowledging the Request.

TC2(Server) The estimated round trip delay between the client and the server. Set when retransmitting after receiving no Response for TC1(Server) time and retransmitting the Request with the APG bit set.

TC3(Server) The estimated maximum expected interpacket time for multi-packet Responses from the Server. Set when waiting for subsequent Response packets within a packet group before timing out.

TC4 The time to wait for additional Responses to a group Request after the first Response is received. This is specified by the user level.

These values are selected as follows. TC1 can be set to TC2 plus a constant, reflecting the time within which most servers respond to most requests. For example, various measurements of VMTP usage at Stanford indicate that 90 percent of the servers respond in less than 200 milliseconds. Setting TC1 to TC2 + 200 means that most Requests receive a Response before timing out and also that overhead for retransmission

for long running transactions is insignificant. A sophisticated implementation may make the estimation of TC1 further specific to the Server.

TC2 may be estimated by measuring the time from when a Probe request is sent to the Server to when a response is received. TC2 can also be measured as the time between the transmission of a Request with the APG bit set to receipt of a management operation acknowledging receipt of the Request.

When the Server is an entity group, TC1 and TC2 should be the largest of the values for the members of the group that are expected to respond. This information may be determined by probing the group on first use (and using the values for the last responses to arrive). Alternatively, one can resort to default values.

TC3 is set initially to 10 times the transmission time for the maximum transmission unit (MTU) to be used for the Response. A sophisticated implementation may record TC3 per Server and refine the estimate based on measurements of actual interpacket gaps. However, a tighter estimate of TC3 only improves the reaction time when a packet is lost in a packet group, at some cost in unnecessary retransmissions when the estimate becomes overly tight.

The server timer, one per active Client, takes on the following values:

TS1(Client) The estimated maximum expected interpacket time. Set when waiting for subsequent Request packets within a packet group before timing out.

TS2(Client) The time to wait to hear from a client before terminating the server processing of a Request. This limits the time spent processing orphan calls, as well as limiting how out of date the server's record of the Client state can be. In particular, TS2 should be significantly less than the minimum time within which it is reasonable to reuse a transaction identifier.

TS3(Client) Estimated roundtrip time to the Client,

TS4(Client) The time to wait after sending a Response (or last hearing from a client) before discarding the state associated with the Request which allows it to filter duplicate Request packets and regenerate the Response.

TS5(Client) The time to wait for an acknowledgment after sending a Response before retransmitting the Response, or giving

up (after some number of retransmissions).

TS1 is set the same as TC3.

The suggested value for TS2 is TC1 + 3*TC2 for this server, giving the Client time to timeout waiting for a Response and retransmit 3 Request packets, asking for acknowledgments.

TS3 is estimated the same as TC1 except that refinements to the estimate use measurements of the Response-to-acknowledgment times.

In the general case, TS4 is set large enough so that a Client issuing a series of closely-spaced Requests to the same Server reuses the same state record at the Server end and thus does not incur the overhead of recreating this state. (The Server can recreate the state for a Client by performing a Probe on the Client to get the needed information.) It should also be set low enough so that the transaction identifier cannot wrap around and so that the Server does not run out of CSR's. We suggest a value in the range of 500 milliseconds. However, if the Server accepts non-idempotent Requests from this Client without doing a Probe on the Client, the TS4 value for this CSR is set to at least 4 times the maximum packet lifetime.

TS5 is TS3 plus the expected time for transmission and reception of the Response. We suggest that the latter be calculated as 3 times the transmission time for the Response data, allowing time for reception, processing and transmission of an acknowledgment at the Client end. A sophisticated implementation may refine this estimate further over time by timing acknowledgments to Responses.

2.5.6. Rate Control

VMTP is designed to deal with the present and future problem of packet overruns. We expect overruns to be the major cause of dropped packets in the future. A client is expected to estimate and adjust the interpacket gap times so as to not overrun a server or intermediate nodes. The selective retransmission mechanism allows the server to indicate that it is being overrun (or some intermediate point is being overrun). For example, if the server requests retransmission of every Kth block, the client should assume overrun is taking place and increase the interpacket gap times. The client passes the server an indication of the interpacket gap desired for a response. The client may have to increase the interval because packets are being dropped by an intermediate gateway or bridge, even though it can handle a higher rate. A conservative policy is to increase the interpacket gap whenever a packet is lost as part of a multi-packet packet group.

The provision of selective retransmission allows the rate of the client and the server to "push up" against the maximum rate (and thus lose packets) without significant penalty. That is, every time that packet transmission exceeds the rate of the channel or receiver, the recovery cost to retransmit the dropped packets is generally far less than retransmitting from the first dropped packet.

The interpacket gap is expressed in 1/32nd's of the MTU packet transmission time. The minimum interpacket gap is 0 and the maximum gap that can be described in the protocol is 8 packet times. This places a limit on the slowest receivers that can be efficiently used on a network, at least those handling multi-packet Requests and Responses. This scheme also limits the granularity of adjustment. However, the granularity is relative to the speed of the network, as opposed to an absolute time. For entities on different networks of significantly different speed, we assume the interconnecting gateways can buffer packets to compensate<2>. With different network speeds and intermediary nodes subject to packet loss, a node must adjust the interpacket gap based on packet loss. The interpacket gap parameter may be of limited use.

2.6. Security

VMTP provides an (optional) secure mode that protects against the usual security threats of peeking, impostoring, message tampering and replays. Secure VMTP must be used to guarantee any of the transport-level reliability properties unless it is guaranteed that there are no intruders or agents that can modify packets and update the packet checksums. That is, non-secure VMTP provides no guarantees in the presence of an intelligent intruder.

The design closely follows that described by Birrell [1]. Authenticated information about a remote entity, including an encryption/decryption key, is obtained and maintained using a VMTP management operation, the authenticated Probe operation, which is executed as a non-secure VMTP message transaction. If a server receives a secure Request for which the server has no entity state, it sends a Probe request to the VMTP

<2> Gateways must also employ techniques to preserve or intelligently modify (if appropriate) the interpacket gaps. In particular, they must be sure not to arbitrarily remove interpacket gaps as a result of their forwarding of packets.

management module of the client, "challenging" it to provide an authenticator that both authenticates the client as being associated with a particular principal as well as providing a key for encryption/decryption. The principal can include a real and effective principal, as used in UNIX <3>. Namely, the real principal is the principal on whose behalf the Request is being performed whereas the effective principal is the principal of the module invoking the request or remote procedure call.

Peeking is prevented by encrypting every Request and Response packet with a working Key that is shared between Client and Server. Impostoring and replays are detected by comparing the Transaction identifier with that stored in the corresponding entity state record (which is created and updated by VMTP as needed). Message tampering is detected by encryption of the packet including the Checksum field. An intruder cannot update the checksum after modifying the packet without knowing the Key. The cost of fully encrypting a packet is close to the cost of generating a cryptographic checksum (and of course, encryption is needed in the general case), so there is no explicit provision for cryptographic checksum without packet encryption.

A Client determines the Principal of the Server and acquires an authenticator for this Server and Principal using a higher level protocol. The Server cannot decrypt the authenticator or the Request packets unless it is in fact the Principal expected by the Client.

An encrypted VMTP packet is flagged by the EPG bit in the VMTP packet header. Thus, encrypted packets are easily detected and demultiplexed from unencrypted packets. An encrypted VMTP packet is entirely encrypted except for the Client, Version, Domain, Length and Packet Flags fields at the beginning of the packet. Client identifiers can be assigned, changed and used to have no real meaning to an intruder or to only communicate public information (such as the host Internet address). They are otherwise just a random means of identification and demultiplexing and do not therefore divulge any sensitive information. Further secure measures must be taken at the network or data link levels if this information or traffic behavior is considered sensitive.

VMTP provides multiple authentication domains as well as an encryption qualifier to accommodate different encryption algorithms and their

---

<3> Principal group membership must be obtained, if needed, by a higher level protocol.

corresponding security/performance trade-offs. (See Appendix V.) A separate key distribution and authentication protocol is required to handle generation and distribution of authenticators and keys. This protocol can be implemented on top of VMTP and can closely follow the Birrell design as well.

Security is optional in the sense that messages may be secure or non-secure, even between consecutive message transactions from the same client. It is also optional in that VMTP clients and servers are not required to implement secure VMTP (although they are required to respond intelligently to attempts to use secure VMTP). At worst, a Client may fail to communicate with a Server if the Server insists on secure communication and the Client does not implement security or vice versa. However, a failure to communicate in this case is necessary from a security standpoint.

2.7. Multicast

The Server entity identifier in a message transaction can identify an entity group, in which case the Request is multicast to every Entity in this group (on a best-efforts basis). The Request is retransmitted until at least one Response is received (or an error timeout occurs) unless it is a datagram Request. The Client can receive multiple Responses to the Request.

The VMTP service interface does not directly provide reliable multicast because it is expensive to provide, rarely needed by applications, and can be implemented by applications using the multiple Response feature. However, the protocol itself is adequate for reliable multicast using positive acknowledgments. In particular, a sophisticated Client implementation could maintain a list of members for each entity group of interest and retransmit the Request until acknowledged by all members. No modifications are required to the Server implementations.

VMTP supports a simple form of subgroup addressing. If the CRE bit is set in a Request, the Request is delivered to the subgroup of entities in the Server group that are co-resident with one or more entities in the group (or individual entity) identified by the CoresidentEntity field of the Request. This is commonly used to send to the manager entity for a particular entity, where Server specifies the group of such managers. Co-resident means "using the same VMTP module", and logically on the same network host. In particular, a Probe request can be sent to the particular VMTP management module for an entity by specifying the VMTP management group as the Server and the entity in question as the CoResidentEntity.

As an experimental aspect of the protocol, VMTP supports the Server sending a group Response which is sent to the Client as well as members of the destination group of Servers to which the original Request was sent. The MDG bit indicates whether the Client is a member of this group, allowing the Server module to determine whether separately addressed packet groups are required to send the Response to both the Client and the Server group. Normally, a Server accepts a group Response only if it has received the Request and not yet responded to the Client. Also, the Server must explicitly indicate it wants to accept group Responses. Logically, this facility is analogous to responding to a mail message sent to a distribution list by sending a copy of the Response to the distribution list.

2.8. Real-time Communication

VMTP provides three forms of support for real-time communication, in addition to its standard facilities, which make it applicable to a wide range of real-time applications. First, a priority is transmitted in each Request and Response which governs the priority of its handling. The priority levels are intended to correspond roughly to:

- urgent/emergency.
- important
- normal
- background.

with additional gradations for each level. The interpretation and implementation of these priority levels is otherwise host-specific, e.g. the assignment to host processing priorities.

Second, datagram Requests allow the Client to send a datagram to another entity or entity group using the VMTP naming, transmission and delivery mechanism, but without blocking, retransmissions or acknowledgment. (The client can still request acknowledgment using the APG bit although the Server does not expect missing portions of a multi-packet datagram Request to be retransmitted even if some are not received.) A datagram Request in non-streamed mode supersedes all previous Requests from the same Client. A datagram Request in stream mode is queued (if necessary) after previous datagram Requests on the same stream. (See Section 2.11.)

Finally, VMTP provides several control bit flags to modify the handling of Requests and Responses for real-time requirements. First, the

conditional message delivery (CMD) flag causes a Request to be discarded if the recipient is not waiting for it when it arrives, similarly for the Response. This option allows a client to send a Request that is contingent on the server being able to process it immediately. The header checksum only (HCO) flag indicates that the checksum has been calculated only on the VMTP header and not on the data segment. Applications such as voice and video can avoid the overhead of calculating the checksum on data whose utility is insensitive to typical bit errors without losing protection on the header information. Finally, the No Retransmission (NRT) flag indicates that the recipient of a message should not ask for retransmission if part of the message is missing but rather either use what was received or discard it.

None of these facilities introduce new protocol states. In fact, the total processing overhead in the normal case is a bit flag test for CMD, HCO or NRT plus assignment of priority on packet transmission and reception. (In fact, CMD and NRT are not tested in the normal case.) The additional code complexity is minimal. We feel that the overhead for providing these real-time facilities is minimal and that these facilities are both important and adequate for a wide class of real-time applications.

Several of the normal facilities of VMTP appear useful for real-time applications. First, multicast is useful for distributed, replicated (fault-tolerant) real-time applications, allowing efficient state query and update for (for example) sensors and control state. Second, the DGM or idempotent flag for Responses has some real-time benefits, namely: a Request is redone to get the latest values when the Response is lost, rather than just returning the old values. The desirability of this behavior is illustrated by considering a request for the current time of day. An idempotent handling of this request gives better accuracy in returning the current time in the case that a retransmission is necessary. Finally, the request-response semantics (in the absence of streaming) of each new Request from a Client terminating the previous message transactions from that Client, if any, provides the "most recent is most important" handling of processing that most real-time applications require.

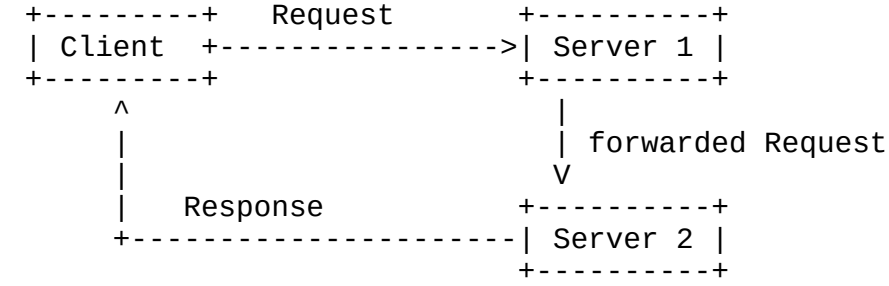
In general, a key design goal of VMTP was provide an efficient general-purpose transport protocol with the features required for real-time communication. Further experience is required to determine whether this goal has been achieved.

2.9. Forwarded Message Transactions

A Server may invoke another Server to handle a Request. It is fairly common for the invocation of the second Server to be the last action performed by the first Server as part of handling the Request. For example, the original Server may function primarily to select a process to handle the Request. Also, the Server may simply check the authorization on the Request. Describing this situation in the context of RPC, a nested remote procedure call may be the last action in the remote procedure and the return parameters are exactly those of the nested call. (This situation is analogous to tail recursion.)

As an optimization to support this case, VMTP provides a Forward operation that allows the server to send the nested Request to the other server and have this other server respond directly to the Client.

If the message transaction being forwarded was not multicast, not secure or the two Servers are the same principal and the ForwardCount of the Request is less than the maximum forward count of 15, the Forward operation is implemented by the Server sending a Request onto the next Server with the forwarded Request identified by the same Client and Transaction as the original Request and a ForwardCount one greater than the Request received from the Client. In this case, the new Server responds directly to the Client. A forwarded Request is illustrated in the following figure.



If the message transaction does not meet the above requirements, the Server's VMTP module issues a nested call and simply maps the returned Response to a Response to original Request without further Server-level processing. In this case, the only optimization over a user-level nested call is one fewer VMTP service operation; the VMTP module handles the return to the invoking call directly. The Server may also use this form of forwarding when the Request is part of a stream of message transactions. Otherwise, it must wait until the forwarded message transaction completes before proceeding with the subsequent message transactions in the stream.

Implementation of the user-level Forward operation is optional, depending on whether the server modules require this facility. Handling an incoming forwarded Request is a minor modification of handling a normal incoming Request. In particular, it is only necessary to examine the ForwardCount field when the Transaction of the Request matches that of the last message transaction received from the Client. Thus, the additional complexity in the VMTP module for the required forwarding support is minimal; the complexity is concentrated in providing a highly optimized user-level Forward primitive, and that is optional.

2.10. VMTP Management

VMTP management includes operations for creating, deleting, modifying and querying VMTP entities and entity groups. VMTP management is logically implemented by a VMTP management server module that is invoked using a message transaction addressed to the Server, VMTP_MANAGER_GROUP, a well-known group entity identifier, in conjunction with Coresident Entity mechanism introduced in Section 2.7. A particular Request may address the local module, the module managing a particular entity, the set of modules managing those entities contained in a specific group or all management modules, as appropriate.

The VMTP management procedures are specified in Appendix III.

2.11. Streamed Message Transactions

Streamed message transactions refer to two or more message transactions initiated by a Client before it receives the response to the first message transaction, with each transaction being processed and responded to in order but asynchronous relative to the initiation of the transactions. A Client streams messages transactions, and thereby has multiple message transactions outstanding, by sending them as part of a single run of message transactions. A run of message transactions is a sequence of message transactions with the same Client and Server and consecutive Transaction identifiers, with all but the first and last Requests and Responses flagged with the NSR (Not Start Run) and NER (Not End Run) control bits. (Conversely, the first Request and Response does not have the NSR set and the last Request and Response does not have the NER bit set.) The message transactions in a run use

consecutive transaction identifiers (except if the STI bit <4> is used in one, in which case the transaction identifier for the next message transaction is 256 greater, rather than 1).

The Client retains a record for each outstanding transaction until it gets a Response or is timed out in error. The record provides the information required to retransmit the Request. On retransmission timeout, the client retransmits the last Request for which it has not received a Response the same as is done with non-streamed communication. (I.e. there need be only one timeout for all the outstanding message transactions associated with a single client.)

The consecutive transaction identifiers within a run of message transactions are used as sequence numbers for error control. The Server handles each message transaction in the sequence specified by its transaction identifier. When it receives a message transaction that is not marked as the beginning of a run, it checks that it previously received a message transaction with the predecessor transaction identifier, either 1 less than the current one or 256 less if the previous one had the STI bit set. If not, the Server sends a NotifyVmtpClient operation to the Client's manager indicating either: (1) the first message transaction was not fully received, or else (2) it has no record of the last one received. If the NRT control flag is set, it does not await nor expect retransmission but proceeds with handling this Request. This flag is used primarily when datagram Requests are used as part of a stream of message transactions. If NRT was not specified, the Client must retransmit from the first message transaction not fully received (either at all or in part) before the Server can proceed with handling this run of Requests or else restart the run of message transactions.

The Client expects to receive the Responses in a consecutive sequence, using the Transaction identifier to detect missing Responses. Thus, the Server must return Responses in sequence except possibly for some gaps, as follows. The Server can specify in the PGcount field in a Response, the number of consecutively previous Responses that this Response

---

<4> The STI bit is used by the Client to effectively allocate 255 transaction identifiers for use by the Server in returning a large Response or stream of Responses.

corresponds to, up to a maximum of 255 previous Responses <5>. Thus, for example, a Response with Transaction identifier 46 and PGcount 3 represents Responses 43, 44, 45 and 46. This facility allows the Server to eliminate sending Responses to Requests that require no Response, effectively batching the Responses into one. It also allows the Server to effectively maintain strictly consecutive sequencing when the Client has skipped 256 Transaction identifiers using the STI bit and the Server does not have that many Responses to return.

If the Client receives a Response that is not consecutive, it retransmits the Request(s) for which the Response(s) is/are missing (unless, of course, the corresponding Requests were sent as datagrams). The Client should wait at the end of a run of message transactions for the last one to complete.

When a Server receives a Request with the NSR bit clear and a higher transaction identifier than it currently has for the Client, it terminates all processing and discards Responses associated with the previous Requests. Thus, a stream of message transactions is effectively aborted by starting a new run, even if the Server was in the middle of handling the previous run.

Using a mixture of datagram and normal Requests as part of a stream of message transactions, particularly with the use of the NRT bit, can lead to complex behavior under packet loss. It is recommended that a run of message transactions be all of one type to avoid problems, i.e. all normal or all datagrams. Finally, when a Server forwards a Request that is part of a run, it must suspend further processing of the subsequent Requests until the forwarded Request has been handled, to preserve order of processing. The simplest handling of this situation is to use a real nested call when forwarding with streamed message transactions.

Flow control of streamed message transactions relies on rate control at the Client plus receipt (or non-receipt) of management notify operations indicating the presence of overrunning. A Client must reduce the number of outstanding message transactions at the Server when it receives a NotifyVmtpServer operation with the MSGTRANS_OVERFLOW ResponseCode. The transact parameter indicates the last packet group that was accepted.

<5> PGcount actually corresponds to packet groups which are described in Section 2.13. This (simplified) description is accurate when there is one Request or Response per packet group.

The implementation of multiple outstanding message transactions requires the ability to record, timeout and buffer multiple outstanding message transactions at the Client end as well as the Server end. However, this facility is optional for both the Client and the Server. Client systems with heavy-weight processes and high network access cost are most likely to benefit from this facility. Servers that serve a wide variety of client machines should implement streaming to accommodate these types of clients.

2.12. Fault-Tolerant Applications

One approach to fault-tolerant systems is to maintain a log of all messages sent at each node and replay the messages at a node when the node fails, after restarting it from the last checkpoint <6>. As an experimental facility, VMTP provides a Receive Sequence Number field in the NotifyVmtpClient and NotifyVmtpServer operations as well as the Next Receive Sequence (NRS) flag in the Response packet to allow a sender to log a receive sequence number with each message sent, allowing the packets to be replayed at a recovering node in the same sequence as they were originally received, thereby recovering to the same state as before.

Basically, each sending node maintains a receive sequence number for each receiving node. On sending a Request to a node, it presume that the receive sequence number is one greater than the one it has recorded for that node. If not, the receiving node sends a notify operation indicating the receive sequence number assigned the Request. The NRS in the Response confirms that the Request message was the next receive sequence number, so the sender can detect if it failed to receive the notify operation in the previous case. With Responses, the packets are ordered by the Transaction identifier except for multicast message transactions, in which there may be multiple Responses with the same identification. In this case, NotifyVmtpServer operations are used to provide receive sequence numbers.

This experimental extension of the protocol is focused on support for fault-tolerant real-time distributed systems required in various critical applications. It may be removed or extended, depending on further investigations.

<6> The sender-based logging is being investigated by Willy Zwaenepoel of Rice University.

2.13. Packet Groups

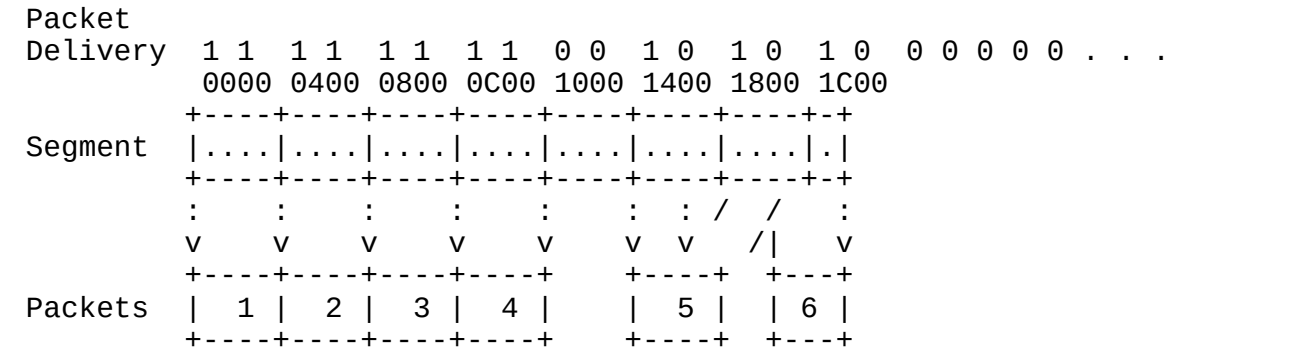
A message (whether Request or Response) is sent as one or more packet groups. A packet group is one or more packets, each containing the same transaction identification and message control block. Each packet is formatted as below with the message control block logically embedded in the VMTP header.

The some fields of the VMTP control portion of the packet and data segment portion can differ between packets within the same packet group.

The segment data portion of a packet group represents up to 16 kilooctets of the segment specified in the message control block. The portion contained in each packet is indicated by the PacketDelivery field contained in the VMTP header. The PacketDelivery field as a bit mask has a similar interpretation to the MsgDelivery field in that each bit corresponds to a segment data block of 512 octets. The PacketDelivery field limits a packet group to 16 kilooctets and a maximum of 32 VMTP packets (with a minimum of 1 packet). Data can be sent in fewer packets by sending multiple data blocks per packet. We require that the underlying datagram service support delivery of (at minimum) the basic 580 octet VMTP packet <7>. To illustrate the use of the PacketDelivery field, consider for example the Ethernet which has a MTU of 1536 octets. so one would send 2 512-octet segment data blocks per packet. (In fact, if a third block is last in the segment and less than 512 octets and fits in the packet without making it too big, an Ethernet packet could contain three data blocks. Thus, an Ethernet packet group for a segment of size 0x1D00 octets (14.5 blocks) and MsgDelivery 0x000074FF consists of 6 packets indicated as follows <8>.

<7> Note that with a 20 octet IP header, a VMTP packet is 600 octets. We propose the convention that any host implementing VMTP implicitly agrees to accept IP/VMTP packets of at least 600 octets.

<8> We use the C notation 0xHHHH to represent a hexadecimal number.



Each '.' is 256 octets of data. The PacketDelivery masks for the 6 packets are: 0x00000003, 0x0000000C, 0x00000030, 0x000000C0, 0x00001400 and 0x00006000, indicating the segment blocks contained in each of the packets. (Note that the delivery bits are in little endian order.)

A packet group is sent as a single "blast" of packets with no explicit flow control. However, the sender should estimate and transmit at a rate of packet transmission to avoid congesting the network or overwhelming the receiver, as described in Section 2.5.6. Packets in a packet group can be sent in any order with no change in semantics.

When the first packet of a packet group is received (assuming the Server does not decide to discard the packet group), the Server saves a copy of the VMTP packet header, indicates it is currently receiving a packet group, initializes a "current delivery mask" (indicating the data in the segment received so far) to 0, accepts this packet (updating the current delivery mask) and sets the timer for the packet group. Subsequent packets in the packet group update the current delivery mask.

Reception of a packet group is terminated when either the current delivery mask indicates that all the packets in the packet group have been received or the packet group reception timer expires (set to TC3 or TS1). If the packet group reception timer expires, if the NRT bit is set in the Control flags then the packet group is discarded if not complete unless MDM is set. In this case, the MsgDelivery field in the message control block is set to indicate the segment data blocks actually received and the message control block and segment data received is delivered to application level.

If NRT is not set and not all data blocks have been received, a NotifyVmtpClient (if a Request) or NotifyVmtpServer (if a Response) is sent back with a PacketDelivery field indicating the blocks received. The source of the packet group is then expected to retransmit the missing blocks. If not all blocks of a Request are received after RequestAckRetries(Client) retransmissions, the Request is discarded and

a NotifyVmtpClient operation with an error response code is sent to the client's manager unless MDM is set. With a Response, there are ResponseAckRetries(Server) retransmissions and then, if MDM is not set, the requesting entity is returned the message control block with an indication of the amount of segment data received extending contiguously from the start of the segment. E.g. if the sender sent 6 512-octet blocks and only the first two and the last two arrived, the receiver would be told that 1024 octets were received. The ResponseCode field is set to BAD_REPLY_SEGMENT. (Note that VMTP is only able to indicate the specific segment blocks received if MDM is set.)

The parameters RequestAckRetries(Client) and ResponseAckRetries(Server) could be set on a per-client and per-server basis in a sophisticated implementation based on knowledge of packet loss.

If the APG flag is set, a NotifyVmtpClient or NotifyVmtpServer operation is sent back at the end of the packet group reception, depending on whether it is a Request or a Response.

At minimum, a Server should check that each packet in the packet group contains the same Client, Server, Transaction identifier and SegmentSize fields. It is a protocol error for any field other than the Checksum, packet group control flags, Length and PacketDelivery in the VMTP header to differ between any two packets in one packet group. A packet group containing a protocol error of this nature should be discarded.

Notify operations should be sent (or invoked) in the manager whenever there is a problem with a unicast packet. i.e. negative acknowledgments are always sent in this case. In the case of problems with multicast packets, the default is to send nothing in response to an error condition unless there is some clear reason why no other node can respond positively. For example, the packet might be a Probe for an entity that is known to have been recently existing on the receiving host but now invalid and could not have migrated. In this case, the receiving host responds to the Probe indicating the entity is nonexistent, knowing that no other host can respond to the Probe. For packets and packet groups that are received and processed without problems, a Notify operation is invoked only if the APG bit is set.

2.14. Runs of Packet Groups

A run of packet groups is a sequence of packet groups, all Request packets or all Response packets, with the same Client and consecutive transaction identifiers, all but the first and last packets flagged with the NSR (Not Start Run) and NER (Not End Run) control bits. When each packet group in the run corresponds to a single Request or Response, it

is identical to a run of message transactions. (See Section 2.11) However, a Request message or a Response message may consists of up to 256 packet groups within a run, for a maximum of 4 megaoctets of segment data. A message that is continued in the next packet group in the run is flagged in the current packet group by the CMG flag. Otherwise, the next packet group in the run (if any) is treated as a separate Request or Response.

Normally, each Request and Response message is sent as a single packet group and each run consists of a single packet group. In this case neither NSR or NER are set. For multi-packet group messages, the PacketDelivery mask in the i-th packet group of a message corresponds to the portion of the segment offset by i-1 times 16 kilooctets, designating the the first packet group to have i = 1.

2.15. Byte Order

For purposes of transmission and reception, the MCB is treated as consisting of 8 32-bit fields and the segment is a sequence of bytes. VMTP transmits the MCB in big-endian order, performing byte-swapping, if necessary, before transmission. A little-endian host must byte-swap the MCB on reception. (The data segment is transmitted as a sequence of bytes with no reordering.) The byte order of the sender of a message is indicated by the LEE bit in the entity identifier for the sender, the Client field if a Request and the Server field if a Response. The sender and receiver of a message are required to agree in some higher level protocol (such as an RPC presentation protocol) on who does further swapping of the MCB and data segment if required by the types of the data actually being transmitted. For example, the segment data may contain a record with 8-bit, 16-bit and 32-bit fields, so additional transformation is required to move the segment from a host of one byte order to another.

VMTP to date has used a higher-level presentation protocol in which segment data is sent in the native order of the sending host and byte-swapped as necessary by the receiving host. This approach minimizes the byte-swapping overhead between machines of common byte order (including when the communication is transparently local to one host), avoids a strong bias in the protocol to one byte-order, and allows for the sending entity to be sending to a group of hosts with different byte orders. (Note that the byte-swap overhead for the MCB is minimal.) The presentation-level overhead is minimal because most common operations, such as file access operations, have parameters that fit the MCB and data segment data types exactly.

2.16. Minimal VMTP Implementation

A minimal VMTP client needs to be able to send a Request packet group and receive a Response packet group as well as accept and respond to Requests sent to its management module, including Probe and NotifyClient operations. It may also require the ability to invoke Probe and Notify operations to locate a Server and acknowledge responses. (the latter only if it is involved in transactions that are not idempotent or datagram message transactions. However, a simple sensor, for example, can transmit VMTP datagram Requests indicating its current state with even less mechanism.) The minimal client thus requires very little code and is suitable as a basis for (e.g.) a network boot loader.

A minimal VMTP server implements idempotent, non-encrypted message transactions, possibly with no segment data support. It should use an entity state record for each Request but need only retain it while processing the Request. Without segment data larger than a packet, there is no need for any timers, buffering (outside of immediate request processing) or queuing. In particular, it needs only as many records as message transactions it handles simultaneously (e.g. 1). The entity state record is required to recognize and respond to Request retransmissions during request processing.

The minimal server need only receive Requests and and be able to send Response packets. It need have only a minimal management module supporting Probe operations. (Support for the NotifyVmtpClient operation is only required if it does not respond immediately to a Request.) Thus the VMTP support for say a time server, sensor, or actuator can be extremely simple. Note that the server need never issue a Probe operation if it uses the host address of the Request for the Response and does not require the Client information returned by the Probe operation. The minimal server should also support reception of forwarded Requests.

2.17. Message vs. Procedural Request Handling

A request-response protocol can be used to implement two forms of semantics on reception. With procedural handling of a Request, a Request is handled by a process associated with the Server that effectively takes on the identity of the calling process, treating the Request message as invoking a procedure, and relinquishing its association to the calling process on return. VMTP supports multiple nested calls spanning multiple machines. In this case, the distributed call stack that results is associated with a single process from the standpoint of authentication and resource management, using the ProcessId field supported by VMTP. The entity identifiers effectively

link these call frames together. That is, the Client field in a Request is effectively the return link to the previous call frame.

With message handling of a Request, a Request message is queued for a server process. The server process dequeues, reads, processes and responds to the Request message, executing as a separate process. Subsequent Requests to the same server are queued until the server asks to receive the next Request.

Procedural semantics have the advantage of allowing each Request (up to the resource limits of the Server) to execute concurrently at the Server, with Request-specific synchronization. Message semantics have the advantage that Requests are serialized at the Server and that the request processing logically executes with the priority, protection and independent execution of a separate process. Note that procedural and message handling of a request appear no differently to the client invoking the message transaction, except possibly for differences in performance.

We view the two Request handling approaches as appropriate under different circumstances. VMTP supports both models.

2.18. Bibliography

The basic protocol is similar to that used in the original form of the V kernel [3, 4] as well as the transport protocol of Birrell and Nelson's [2] remote procedure call mechanism. An earlier version of the protocol was described in SIGCOMM'86 [6]. The rate-based flow control is similar to the techniques of Netblt [9]. The support for idempotency draws, in part, on the favorable experience with idempotency in the V distributed system. Its use was originally inspired by the Woodstock File Server [11]. The multicast support draws on the multicast facilities in V [5] and is designed to work with, and is now implemented using, the multicast extensions to the Internet [8] described in RFC 966 and 988. The secure version of the protocol is similar to that described by Birrell [1] for secure RPC. The use of runs of packet groups is similar to Fletcher and Watson's delta-T protocol [10]. The use of "management" operations implemented using VMTP in place of specialized packet types is viewed as part of a general strategy of using recursion to simplify protocol architectures [7].

Finally, this protocol was designed, in part, to respond to the requirements identified by Braden in RFC 955. We believe that VMTP satisfies the requirements stated in RFC 955.

[1] A.D. Birrell, "Secure Communication using Remote Procedure Calls", ACM. Trans. on Computer Systems 3(1), February, 1985.

[2] A. Birrell and B. Nelson, "Implementing Remote Procedure Calls", ACM Trans. on Computer Systems 2(1), February, 1984.

[3] D.R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations", In Proceedings of the 9th Symposium on Operating System Principles, ACM, 1983.

[4] D.R. Cheriton, "The V Kernel: A Software Base for Distributed Systems", IEEE Software 1(2), April, 1984.

[5] D.R. Cheriton and W. Zwaenepoel, "Distributed Process Groups in the V Kernel", ACM Trans. on Computer Systems 3(2), May, 1985.

[6] D.R. Cheriton, "VMTP: A Transport Protocol for the Next Generation of Communication Systems", In Proceedings of SIGCOMM'86, ACM, Aug 5-7, 1986.

[7] D.R. Cheriton, "Exploiting Recursion to Simplify an RPC Communication Architecture", in preparation, 1988.

[8] D.R. Cheriton and S.E. Deering, "Host Groups: A Multicast Extension for Datagram Internetworks", In 9th Data Communication Symposium, IEEE Computer Society and ACM SIGCOMM, September, 1985.

[9] D.D. Clark and M. Lambert and L. Zhang, "NETBLT: A Bulk Data Transfer Protocol", Technical Report RFC 969, Defense Advanced Research Projects Agency, 1985.

[10] J.G. Fletcher and R.W. Watson, "Mechanism for a Reliable Timer-based Protocol", Computer Networks 2:271-290, 1978.

[11] D. Swinehart and G. McDaniel and D. Boggs, "WFS: A Simple File System for a Distributed Environment", In Proc. 7th Symp. Operating Systems Principles, 1979.

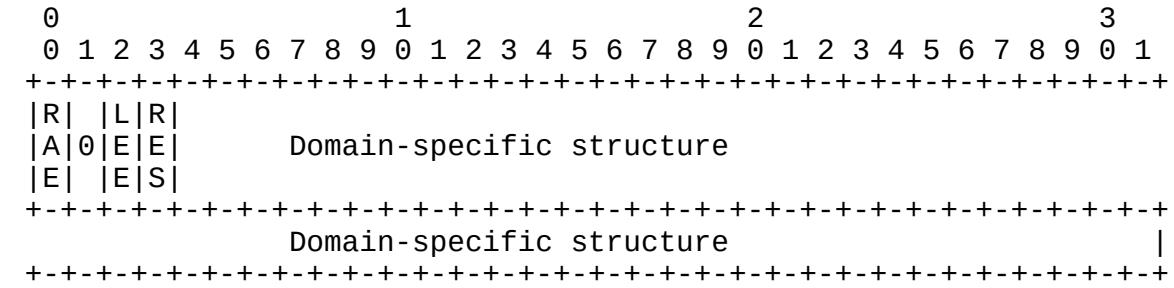
3. VMTP Packet Formats

VMTP uses 2 basic packet formats corresponding to Request packets and Response packets. These packet formats are identical in most of the fields to simplify the implementation.

We first describe the entity identifier format and the packet fields that are used in general, followed by a detailed description of each of the packet formats. These fields are described below in detail. The individual packet formats are described in the following subsections. The reader and VMTP implementor may wish to refer to Chapters 4 and 5 for a description of VMTP event handling and only refer to this detailed description as needed.

3.1. Entity Identifier Format

The 64-bit non-group entity identifiers have the following substructure.



The field meanings are as follows:

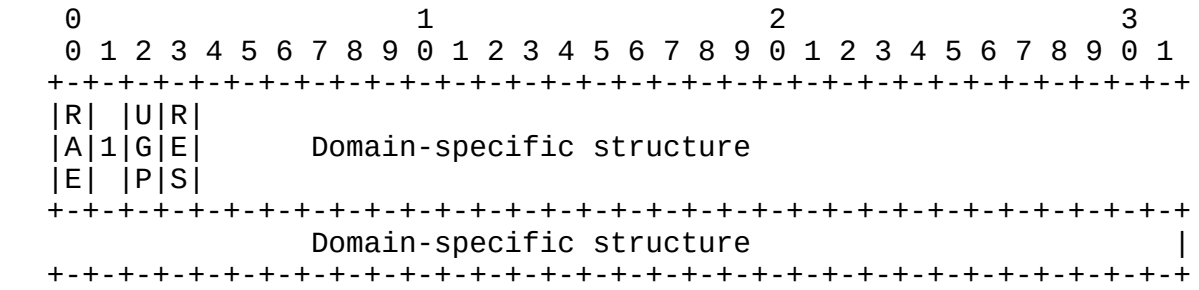
RAE — Remote Alias Entity - the entity identifier identifies an entity that is acting as an alias for some entity outside this entity domain. This bit is used by higher-level protocols. For instance, servers may take extra security and protection measures with aliases.

GRP — Group - 0, for non-group entity identifiers.

LEE — Little-Endian Entity - the entity transmits data in little-endian (VAX) order.

RES — Reserved - must be 0.

The 64-bit entity group identifiers have the following substructure.



The field meanings are as follows:

RAE Remote Alias Entity - same as for non-group entity identifier.

GRP Group - 1, for entity group identifiers.

UGP Unrestricted Group - no restrictions are placed on joining this group. I.e. any entity can join limited only by implementation resources.

RES Reserved - must be 0.

The all-zero entity identifier is reserved and guaranteed to be unallocated in all domains. In addition, a domain may reserve part of the entity identifier space for statically allocated identifiers. However, this is domain-specific.

Description of currently defined entity identifier domains is provided in Appendix IV.

3.2. Packet Fields

Client 64-bit identifier for the client entity associated with this packet. The structure, allocation and binding of this identifier is specific to the specified Domain. An entity identifier always includes 4 types bits as specified in Section 3.1.

Version The 3-bit identifier specifying the version of the protocol. Current version is version 0.

Domain The 13-bit identifier specifying the naming and administration domain for the client and server named in the packet.

Packet Flags: 3 bits. (The normal case has none of the flags set.)

HCO — Header checksum only - checksum has only been calculated on the header. This is used in some real-time applications where the strict correctness of the data is not needed.

EPG — Encrypted packet group - part of a secure message transaction.

MPG — Multicast packet group - packet was multicast on transmission.

Length — A 13-bit field that specifies the number of 32-bit words in the segment data portion of the packet (if any), excluding the checksum field. (Every VMTP packet is required to be a multiple of 64 bits, possibly by padding out the segment data.) The minimum legal Length is 0, the maximum length is 4096 and it must be an even number.

Control Flags: 9 bits. (The normal case has none of the flags set.)

NRS — Next Receive Sequence - the associated Request message (in a Response) or previous Response (if a Request) was received consecutive with the last Request from this entity. That is, there was no interfering messages received.

APG — Acknowledge Packet Group - Acknowledge packet group on receipt. If a Request, send back a Request to the client’s manager providing an update on the state of the transaction as soon as the request packet group is received, independent of the response being available. If a Response, send an update to the server’s manager as soon as possible after response packet group is received providing an update on the state of the transaction at the client

NSR — Not Start Run - 1 if this packet is not part of the first packet group of a run of packet groups.

NER — Not End Run - 1 if this packet is not part of the last packet group of a run of packet groups.

NRT — No Retransmission - do not ask for retransmissions of this packet group if not all received within timeout

period, just deliver or discard.

MDG Member of Destination Group - this packet is sent to a group and the client is a member of this group.

CMG Continued Message - the message (Request or Response) is continued in the next packet group. The next packet group has to be part of the same run of packet groups.

STI Skip Transaction Identifiers - the next transaction identifier that the Client plans to use is the current transaction plus 256, if part of the same run and at least this big if not. In a Request, this authorizes the Server to send back up to 256 packet groups containing the Response.

DRT Delay Response Transmission - set by request sender if multiple responses are expected (as indicated by the MRD flag in the RequestCode) and it may be overrun by multiple responses. The responder(s) should then introduce a short random delay in sending the Response to minimize the danger of overrunning the Client. This is normally only used for responding to multicast Requests where the Client may be receiving a large number of Responses, as indicated by the MRD flag in the Request flags. Otherwise, the Response is sent immediately.

RetransmitCount:
3 bits - the ordinal number of transmissions of this packet group prior to this one, modulo 8. This field is used in estimation of roundtrip times. This count may wrap around during a message transaction. However, it should be sufficient to match acknowledgments and responses with a particular transmission.

ForwardCount: 4 bits indicating the number of times this Request has been forwarded. The original Request is always sent with a ForwardCount of 0.

Interpacket Gap: 8 bits.
Indicates the recommended time to use between subsequent packet transmissions within a multi-packet packet group transmission. The Interpacket Gap time is in 1/32nd of a network packet transmission time for a packet of size MTU for the node. (Thus, the maximum gap time is 8 packet times.)

PGcount: 8 bits
The number of packet groups that this packet group represents in addition to that specified by the Transaction field. This is used in acknowledging multiple packet groups in streamed communication.

Priority 4-bit identifier for priority for the processing of this request both on transmission and reception. The interpretation is:

1100	urgent/emergency
1000	important
0000	normal
0100	background

Viewing the higher-order bit as a sign bit (with 1 meaning negative), low values are high priority and high values are low priority. The low-order 2 bits indicate additional (lower) gradations for each level.

Function Code: 1 bit - types of VMTP packets. If the low-order bit of the function code is 0, the packet is sent to the Server, else it is sent to the Client.

0	Request
1	Response

Transaction: 32 bits:
Identifier for this message transaction.

PacketDelivery: 32 bits:
Delivery indicates the segment blocks contained in this packet. Each bit corresponds to one 512-octet block of segment data. A 1 bit in the i-th bit position (counting the LSB as 0) indicates the presence of the i-th segment block.

Server: 64 bits
Entity identifier for the server or server group associated with this transaction. This is the receiver when a Request packet and the sender when a Response packet.

Code: 32 bits The Request Code and Response Code, set either at the user level or VMTP level depending on use and packet type. Both the Request and Response codes include 8 high-order bits from the following set of control bits:

CMD Conditional Message Delivery - only deliver the request or response if the receiving entity is waiting for it at the time of delivery, otherwise drop the message.

DGM DataGram Message - indicates that the message is being sent as a datagram. If a Request message, do not wait for reply, or retransmit. If a Response message, treat this message transaction as idempotent.

MDM Message Delivery Mask - indicates that the MsgDelivery field is being used. Otherwise, the MsgDelivery field is available for general use.

SDA Segment Data Appended - segment data is appended to the message control block, with the total size of the segment specified by the SegmentSize field. Otherwise, the segment data is null and the SegmentSize field is not used by VMTP and available for user- or RPC-level uses.

CRE CoResident Entity - indicates that the CoResidentEntity field in the message should be interpreted by VMTP. Otherwise, this field is available for additional user data.

MRD Multiple Responses Desired - multiple Responses are desired to to this Request if it is multicast. Otherwise, the VMTP module can discard subsequent Responses after the first Response.

PIC Public Interface Code - Values for Code with this bit set are reserved for definition by the VMTP specification and other standard protocols defined on top of VMTP.

RES Reserved for future use. Must be 0.

CoResidentEntity
64-bit Identifier for an entity or group of entities with which the Server entity or entities must be co-resident, i.e. route only to entities (identified by Server) on the same host(s) as that specified by

CoResidentEntity, Only meaningful if CRE is set in the Code field.

User Data 12 octets Space in the header for the VMTP user to specify user-specific control and data.

MsgDelivery: 32 bits
The segment blocks being transmitted (in total) in this packet group following the conventions for the PacketDelivery field. This field is ignored by the protocol and treated as an additional user data field if MDM is 0. On transmission, the user level sets the MsgDelivery to indicate those portions of the segment to be transmitted. On receipt, the MsgDelivery field is modified by the VMTP module to indicate the segment data blocks that were actually received before the message control block is passed to the user or RPC level. In particular, the kernel does not discard the packet group if segment data blocks are missing. A Server or Client entity receiving a message with a MsgDelivery in use must check the field to ensure adequate delivery and retry the operation if necessary.

SegmentSize: 32 bits
Size of segment in octets, up to a maximum of 16 kilooctets without streaming and 4 megaoctets with streaming, if SDA is set. Otherwise, this field is ignored by the protocol and treated as an additional user data field.

Segment Data: 0-16 kilooctets
0 octets if SDA is 0, else the portion of the segment corresponding to the Delivery Mask, limited by the SegmentSize and the MTU, padded out to a multiple of 64 bits.

Checksum: 32 bits.
The 32-bit checksum for the header and segment data.

The VMTP checksum algorithm <9> develops a 32-bit checksum by computing

---

<9> This algorithm and description are largely due to Steve Deering of Stanford University.

two 16-bit, ones-complement sums (like IP), each covering different parts of the packet. The packet is divided into clusters of 16 16-bit words. The first, third, fifth,... clusters are added to the first sum, and the second, fourth, sixth,... clusters are added to the second sum. Addition stops at the end of the packet; there is no need to pad out to a cluster boundary (although it is necessary that the packet be an integral multiple of 64 bits; padding octets may have any value and are included in the checksum and in the transmitted packet). If either of the resulting sums is zero, it is changed to 0xFFFF. The two sums are appended to the transmitted packet, with the first sum being transmitted first. Four bytes of zero in place of the checksum may be used to indicate that no checksum was computed.

The 16-bit, ones-complement addition in this algorithm is the same as used in IP and, therefore, subject to the same optimizations. In particular, the words may be added up 32-bits at a time as long as the carry-out of each addition is added to the sum on the following addition, using an "add-with-carry" type of instruction. (64-bit or 128-bit additions would also work on machines that have registers that big.)

A particular weakness of this algorithm (shared by IP) is that it does not detect the erroneous swapping of 16-bit words, which may easily occur due to software errors. A future version of VMTP is expected to include a more secure algorithm, but such an algorithm appears to require hardware support for efficient execution.

Not all of these fields are used in every packet. The specific packet formats are described below. If a field is not mentioned in the description of a packet type, its use is assumed to be clear from the above description.

3.3. Request Packet

The Request packet (or packet group) is sent from the client to the server or group of servers to solicit processing plus the return of zero or more responses. A Request packet is identified by a 0 in the LSB of the fourth 32-bit word in the packet.

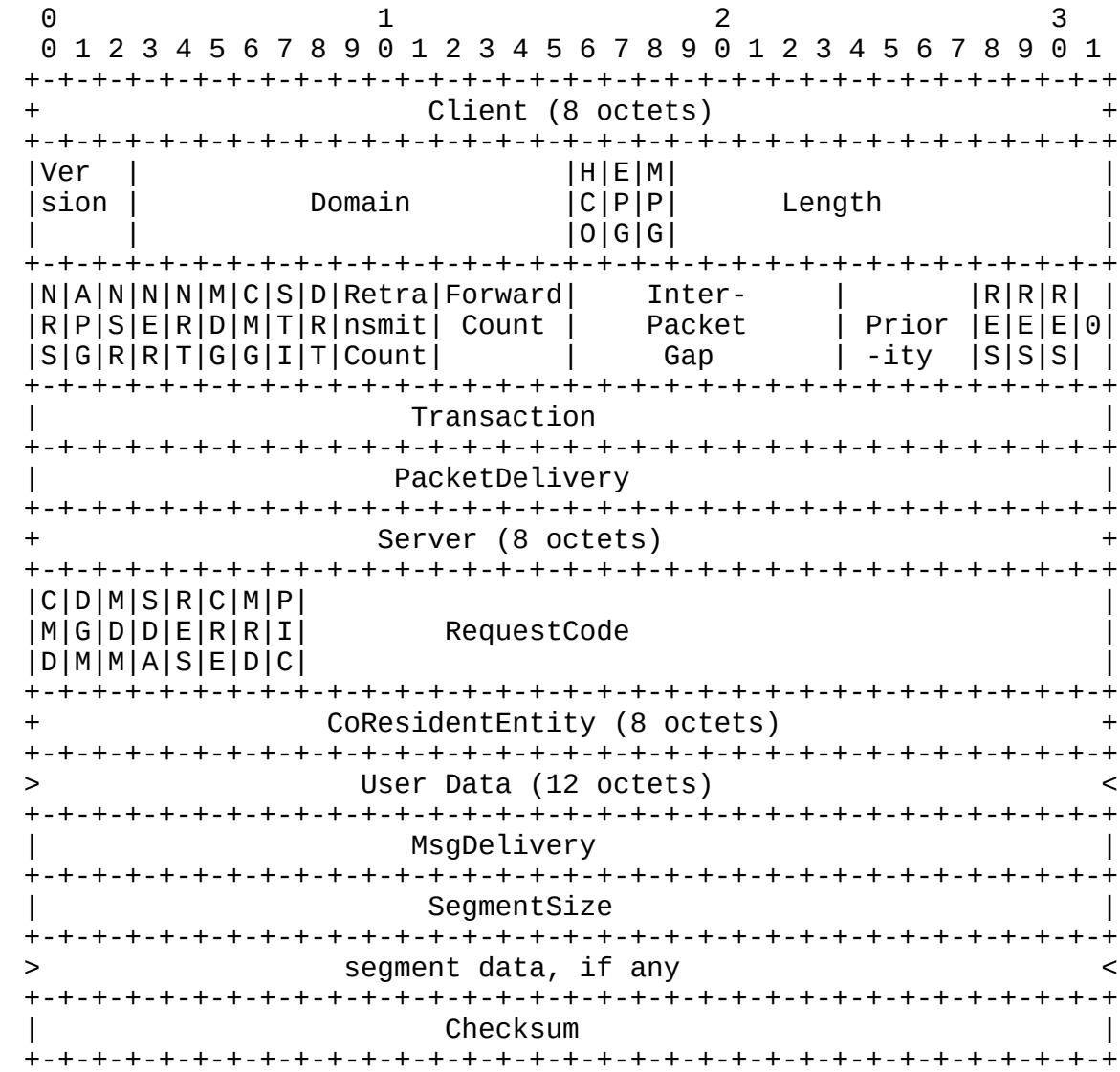


Figure 3-1: Request Packet Format

The fields of the Request packet are set according to the semantics described in Section 3.2 with the following qualifications.

InterPacketGap	The estimated interpacket gap time the client would like for the Response packet group to be sent by the Server in responding to this Request.
Transaction	Identifier for transaction, at least one greater than the previously issued Request from this Client.
Server	Server to which this Request is destined.
RequestCode	Request code for this request, indicating the operation to perform.

3.4. Response Packet

The Response packet is sent from the Server to the Client in response to a Request, identified by a 1 in the LSB of the fourth 32-bit word in the packet.

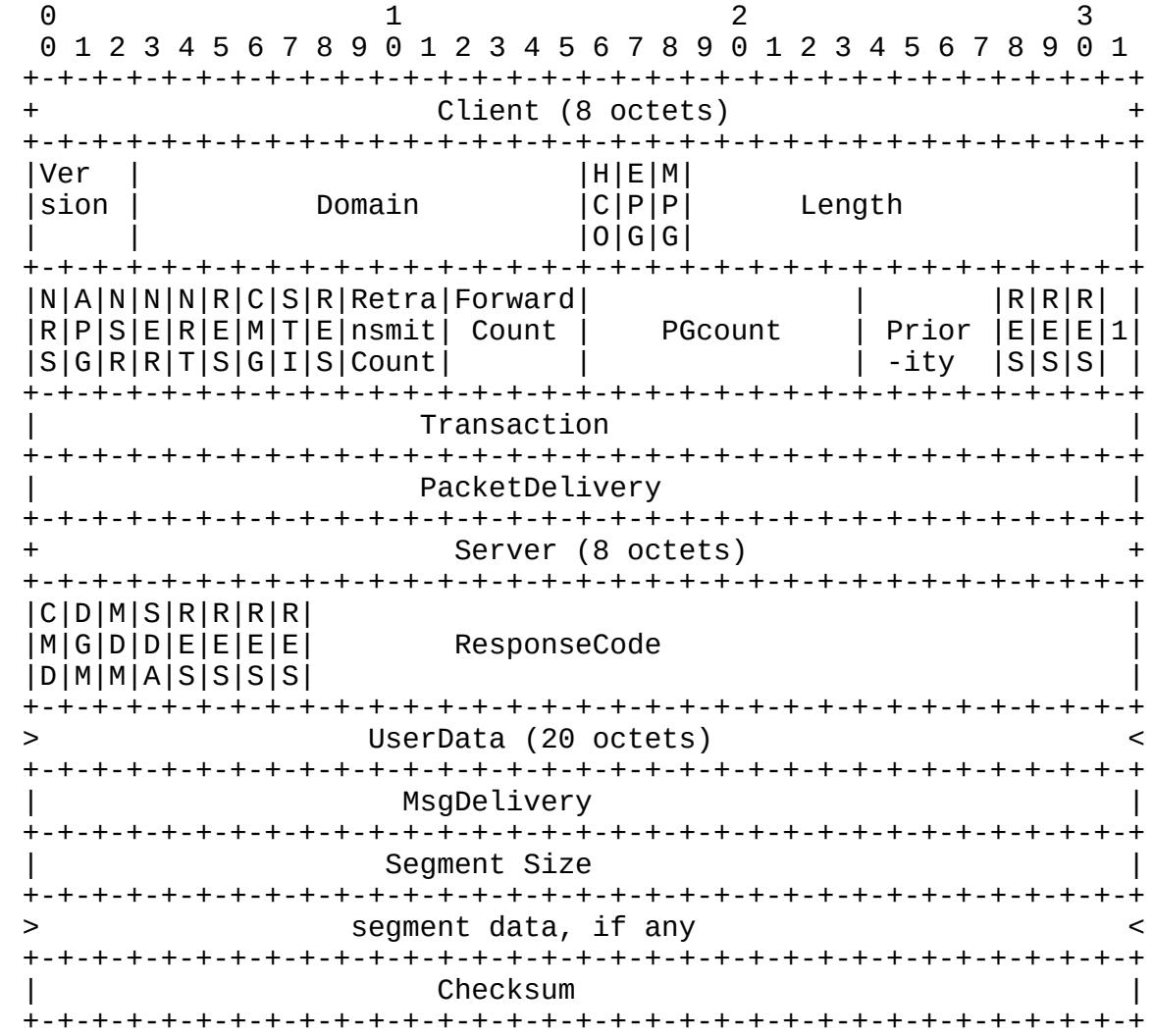


Figure 3-2: Response Packet Format

The fields of the Response packet are set according to the semantics described in Section 3.2 with the following qualifications.

Client, Version, Domain, Transaction
Match those in the Request packet group to which this is

a response.

STI 1 if this Response is using one or more of the transaction identifiers skipped by the Client after the Request to which this is a Response. STI in the Request essentially allocates up to 256 transaction identifiers for the Server to use in a run of Response packet groups.

RetransmitCount The retransmit count from the last Request packet received to which this is a response.

ForwardCount The number of times the corresponding Request was forwarded before this Response was generated.

PGcount The number of consecutively previous packet groups that this response is acknowledging in addition to the one identified by the Transaction identifier.

Server Server sending this response. This may differ from that originally specified in the Request packet if the original Server was a server group, or the request was forwarded.

The next two chapters describes the protocol operation using these packet formats, with the the Client and the Server portions described separately.

4. Client Protocol Operation

This chapter describes the operation of the client portion of VMTP in terms of the procedures for handling VMTP user events, packet reception events, management operations and timeout events. Note that the client portion of VMTP is separable from the server portion. It is feasible to have a node that only implements the client end of VMTP.

To simplify the description, we define a client state record (CSR) plus some standard utility routines.

4.1. Client State Record Fields

In the following protocol description, there is one client state record (CSR) per (client,transaction) outstanding message transaction. Here is a suggested set of fields.

Link — Link to next CSR when queued in one of the transmission, timeout or message queues.

QueuePtr — Pointer to queue head in which this CSR is contained or NULL if none. Queue could be one of transmission queue, timeout queue, server queue or response queue.

ProcessIdentification — The process identification and address space.

Priority — Priority for processing, network service, etc.

State — One of the client states described below.

FinishupFunc — Procedure to be executed on the CSR when it is completes its processing in transmission or timeout queues.

TimeoutCount — Time to remain in timeout queue.

TimeoutLimit — User-specified time after which the message transaction is aborted. The timeout is infinite if set to zero.

RetransCount — Number of retransmissions since last hearing from the Server.

LastTransmitTime — The time at which the last packet was sent. This field is used to calculate roundtrip times, using the RetransmitCount to match the responding packet to a

particular transmission. I.e. Response or management NotifyVmtpClient operation to Request and a management NotifyVmtpServer operation to a Response.

TimetoLive — Time to live to be used on transmission of IP packets.

TransmissionMask — Bit mask indicating the portions of the segment to transmit. Set before entering the transmission queue and cleared incrementally as the 512-byte segment blocks of the segment are transmitted.

LocalClientLink — Link to next CSR hashing to same hash index in the ClientMap.

LocalClient — Entity identifier for client when this CSR is used to send a Request packet.

LocalTransaction — Transaction identifier for current message transaction the local client has outstanding.

LocalPrincipal — Account identification, possibly including key and key timeout.

LocalDelivery — Bit mask of segment blocks that have not been acknowledged in the Request or have been received in the Response, depending on the state.

ResponseQueue — Queue of CSR's representing the queued Responses for this entity.

VMTP Header — Prototype VMTP header, used to generate and store the header portion of a Request for transmission and retransmission on timeout.

SegmentDesc — Description of the segment data associated with the CSR, either the area storing the original Request data, the area for receiving Request data, or the area storing the Response data that is returned.

HostAddr — The network or internetwork host address to which the Client last transmitted. This field also indicates the type of the address, e.g. IP, Ethernet, etc.

Note: the CSR can be combined with a light-weight process descriptor with considerable benefit if the process is designed to block when it

issues a message transaction. In particular, by combining the two descriptors, the implementation saves time because it only needs to locate and queue one descriptor with various operations (rather than having to locate two descriptors). It also saves space, given that the VMTP header prototype provides space such as the user data field which may serve to store processor state for when the process is preempted. Non-preemptive blocking can use the process stack to store the processor state so only a program counter and stack pointer may be required in the process descriptor beyond what we have described. (This is the approach used in the V kernel.)

4.2. Client Protocol States

A Client State Record records the state of message transaction generated by this host, identified by the (Client, Transaction) values in the CSR. As a client originating a transaction, it is in one of the following states.

AwaitingResponse
: Waiting for a Response packet group to arrive with the same (Client,Transaction) identification.

ReceivingResponse
: Waiting for additional packets in the Response packet group it is currently receiving.

"Other"
: Not waiting for a response, which can be Processing or some other operating system state, or one of the Server states if it also acts as a server.

This covers all the states for a client.

4.3. State Transition Diagrams

The client state transitions are illustrated in Figure 4-1. The client goes into the state AwaitingResponse on sending a request unless it is a datagram request. In the AwaitingResponse state, it can timeout and retry and eventually give up and return to the processing state unless it receives a Response. (A NotifyVmtpClient operation resets the timeout but does not change the state.) On receipt of a single packet response, it returns to the processing state. Otherwise, it goes to ReceivingResponse state. After timeout or final response packet is received, the client returns to the processing state. The processing state also includes any other state besides those associated with issuing a message transaction.

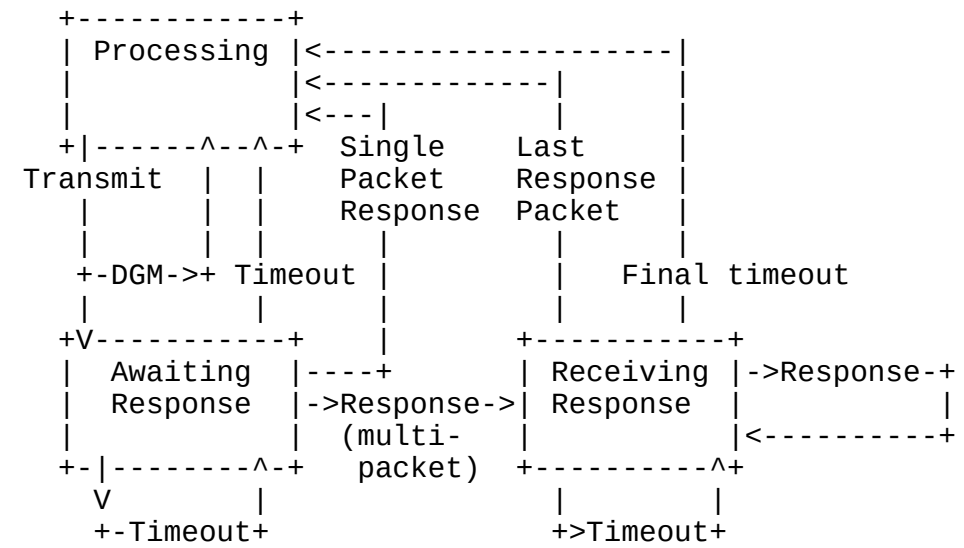


Figure 4-1: Client State Transitions

4.4. User Interface

The RPC or user interface to VMTP is implementation-dependent and may use systems calls, functions or some other mechanism. The list of requests that follow is intended to suggest the basic functionality that should be available.

Send(mcb, timeout, segptr, segsize)
: Initiate a message transaction to the server and request message specified by mcb and return a response in mcb, if it is received within the specified timeout period (or else return USER_TIMEOUT in the Code field). The segptr parameter specifies the location from which the segment data is sent and the location into which the response data is to be delivered. The segsize field indicates the maximum length of this area.

GetResponse(responsemcb, timeout, segptr, segsize)
: Get the next response sent to this client as part of the current message transaction, returning the segment data, if any, into the memory specified by segptr and segsize.

This interface assumes that there is a client entity associated with the invoking process that is to be used with these operations. Otherwise, the client entity must be specified as an additional parameter.

4.5. Event Processing

The following events may occur in the VMTP client:

- User Requests
    - Send
    - GetResponse
- Packet Arrival
    - Response Packet
    - Request

  The minimal Client implementation handles Request packets for its VMTP management (server) module and sends NotifyVmtpClient requests in response to others, indicating the specified server does not exist.

- Management Operation - NotifyVmtpClient
- Timeouts
    - Client Retransmission Timeout

The handling of these events is described in detail in the following subsections.

We first describe some conventions and procedures used in the description. A field of the received packet is indicated as (for example) p.Transaction, for the Transaction field. Optional portions of the code, such as the streaming handling code are prefixed with a "|" in the first column.

```
MapClient( client )
                Return pointer to CSR for client with the specified
                clientId, else NULL.

SendPacketGroup( csr )
                Send the packet group (Request, Response) according to
                that specified by the CSR.

NotifyClient( csr, p, code )
                Invoke the NotifyVmtpClient operation with the
                parameters csr.RemoteClient, p.control,
```

csr.ReceiveSeqNumber, csr.RemoteTransaction and csr.RemoteDelivery, and code. If csr is NULL, use p.Client, p.Transaction and p.PacketDelivery instead and the global ReceiveSequenceNumber, if supported. This function simplifies the description over calling NotifyVmtpClient directly in the procedural specification below. (See Appendix III.)

NotifyServer(csr, p, code)
Invoke the NotifyVmtpServer operation with the parameters p.Server, csr.LocalClient, csr.LocalTransaction, csr.LocalDelivery and code. Use p.Client, P.Transaction and 0 for the clientId, transact and delivery parameters if csr is NULL. This function simplifies the description over calling NotifyVmtpServer directly in the procedural specification below. (See Appendix III.)

DGMset(p) True if DGM bit set in packet (or csr) else False. (Similar functions are used for other bits.)

Timeout(csr, timeperiod, func)
Set or reset timer on csr record for timeperiod later and invoke func if the timeout expires.

4.6. Client User-invoked Events

A user event occurs when a VMTP user application invokes one of the VMTP interface procedures.

4.6.1. Send

```
Send( mcb, timeout, segptr, segsize )
    map to main CSR for this client.
    increment csr.LocalTransaction
    Init csr and check parameters and segment if any.
    Set SDA if sending appended data.
    Flush queued replies from previous transaction, if any.
    if local non-group server then
        deliver locally
        await response
        return
    if GroupId(server) then
        Check for and deliver to local members.
        if CRE request and non-group local CR entity then
```

```
            await response
            return
         endif
         set MDG if member of this group.
    endif
    clear csr.RetransCount
    set csr.TransmissionMask
    set csr.TimeLimit to timeout
    set csr.HostAddr for csr.Server
    SendPacketGroup( csr )
    if DGMset(csr) then
       return
    endif
    set csr.State to AwaitingResponse
    Timeout( rootcsr, TC1(csr.Server), LocalClientTimeout )
    return
end Send
```

Notes:

1. Normally, the HostAddr is extracted from the ServerHost cache, which maps server entity identifiers to host addresses. However, on cache miss, the client first queries the network using the ProbeEntity operation, as specified in Appendix III, determining the host address from the Response. The ProbeEntity operation is handled as a separate message transaction by the Client.

The stream interface incorporates a parameter to pass a responseHandler procedure that is invoked when the message transaction completes.

```
StreamSend( mcb, timeout, segptr, segsize, responseHandler )
    map to main CSR for this client.
|   Allocate a new csr if root in use.
|   lastcsr := First csr for last request.
|   if STIset(lastcsr)
|       csr.LocalTransaction := lastcsr.LocalTransaction + 256
|   else
|       csr.LocalTransaction := lastcsr.LocalTransaction + 1
    Init csr and check parameters and segment if any.
    . . . ( rest is the same as for the normal Send)
```

Notes:

1. Each outstanding message transaction is represented by a CSR queued on the root CSR for this client entity. The root CSR is used to handle timeouts, etc. On timeout, the last packet

from the last packet group is retransmitted (with or without the segment data).

4.6.2. GetResponse

```
GetResponse( req, timeout, segptr, segsize )
    csr := CurrentCSR;
    if responses queued then return next response
      (in req, segptr to max of segsize )
    if timeout is zero then return KERNEL_TIMEOUT error
    set state to AWAITING_RESPONSE
    Timeout( csr, timeout, ReturnKernelTimeout );
end GetResponse
```

Notes:

1. GetResponse is only used with multicast Requests, which is the only case in which multiple (different) Responses should be received.

2. A response must remain queued until the next message transaction is invoked to filter out duplicates of this response.

3. If the response is incomplete (only relevant if a multi-packet response), then the client may wait for the response to be fully received, including issuing requests for retransmission (using NotifyVmtpServer operations) before returning the response.

4. As an optimization, a response may be stored in the CSR of the client. In this case, the response must be transferred to a separate buffer (for duplicate suppression) before waiting for another response. Using this optimization, a response buffer is not allocated in the common case of the client receiving only one response.

4.7. Packet Arrival

In general, on packet reception, a packet is mapped to the client state record, decrypted if necessary using the key in the CSR. It then has its checksum verified and then is transformed to the right byte order. The packet is then processed fully relative to its packet function code. It is discarded immediately if it is addressed to a different domain than the domain(s) in which the receiving host participates.

For each of the 2 packet types, we assume a procedure called with a pointer p to the VMTP packet and psize, the size of the packet in octets. Thus, generic packet reception is:

```
if not LocalDomain(p.Domain) then return;

csr := MapClient( p.Client )

if csr is NULL then
    HandleNoCsr( p, psize )
    return

if Secure(p) then
    if SecureVMTP not supported then
        { Assume a Request. }
        if not Multicast(p) then
            NotifyClient(NULL, p, SECURITY_NOT_SUPPORTED )
        return
    endif
|   Decrypt( csr.Key, p, psize )

if p.Checksum not null then
    if not VerifyChecksum(p, psize) then return;
if OppositeByteOrder(p) then ByteSwap( p, psize )
if psize not equal sizeof(VmtpHeader) + 4*p.Length then
    NotifyClient(NULL, p, VMTP_ERROR )
    return
Invoke Procedure[p.FuncCode]( csr, p, psize )
Discard packet and return
```

Notes:

1. The Procedure[p.FuncCode] refers to one of the 2 procedures corresponding to the two different packet types of VMTP, Requests and Responses.

2. In all the following descriptions, a packet is discarded on "return" unless otherwise stated.

3. The procedure HandleNoCSR is a management routine that allocates and initializes a CSR and processes the packet or else sends an error indication to the sender of the packet. This procedure is described in greater detail in Section 4.8.1.

4.7.1. Response

This procedure handles incoming Response packets.

```
HandleResponse( csr, p, psize )
    if not LocalClient( csr ) then
        if Multicast then return
|       if Migrated( p.Client ) then
|           NotifyServer(csr, p ENTITY_MIGRATED )
|       else
            NotifyServer(csr, p, ENTITY_NOT_HERE )
        return
    endif

    if NSRset(p) then
        if Streaming not supported then
            NotifyServer(csr, p, STREAMING_NOT_SUPPORTED )
            return STREAMED_RESPONSE
|       Find csr corresponding to p.Transaction
|       if none found then
|           NotifyServer(csr, p, BAD_TRANSACTION_ID )
|           return
     else
      if csr.LocalTransaction not equal p.Transaction then
        NotifyServer(csr, p, BAD_TRANSACTION_ID )
        return
    endif
    Locate reply buffer rb for this p.Server
    if found then
        if rb.State is not ReceivingResponse then
          { Duplicate }
            if APGset(p) or NERset(p) then
                { Send Response to stop response packets. }
                NotifyServer(csr, p, RESPONSE_DISCARDED )
            endif
            return
         endif
         { rb.State is ReceivingRequest}
         if new segment data then retain in CSR segment area.
         if packetgroup not complete then
             Timeout( rb, TC3(p.Server), LocalClientTimeout )
             return;
          endif
          goto EndPacketGroup
    endif
    { Otherwise, a new response message. }
```

```
    if (NSRset(p) or NERset(p)) and NoStreaming then
        NotifyServer(csr, p, VMTP_ERROR )
        return
|    if NSRset(p) then
|      { Check consecutive with previous packet group }
|       Find last packet group CSR from p.Server.
|       if p.Transaction not
|             lastcsr.RemoteTransaction+1 mod 2**32 then
|         { Out of order packet group }
|            NotifyServer(csr, p, BAD_TRANSACTION_ID)
|           return
|       endif
|       if lastcsr not completed then
|           NotifyServer(lastcsr, p, RETRY )
|       endif
|       if CMG(lastcsr) then
|           Add segment data to lastcsr Response
|           Notify lastcsr with new packet group.
|           Clear lastcsr.VerifyInterval
|       else
|           if lastcsr available then
|                 use it for this packet group
|           else allocate and initialize new CSR
|           Save message and segment data in new CSR area.
|       endif
|   else { First packet group }
        Allocate and init reply buffer rb for this response.
        if allocation fails then
            NotifyServer(csr, p, BUSY )
            return
        Set rb.State to ReceivingResponse
        Copy message and segment data to rb's segment area
         and set rb.PacketDelivery to that delivered.
        Save p.Server host address in ServerHost cache.
    endif
    if packetgroup not complete then
        Timeout( rb, TS1(p.Client), LocalClientTimeout )
        return;
    endif
endPacketGroup:
    { We have received last packet in packet group. }
    if APGset(p) then NotifyServer(csr, p, OK )
|   if NERset(p) and CMGset(p) then
|       Queue waiting for continuation packet group.
|       Timeout( rb, TC2(rb.Server), LocalClientTimeout )
|       return
|   endif
```

```
    { Deliver response message. }
    Deliver response to Client, or queue as appropriate.
end HandleResponse
```

Notes:

1. The mechanism for handling streaming is optional and can be replaced with the tests for use of streaming. Note that the server should never stream at the Client unless the Client has streamed at the Server or has used the STI control bit. Otherwise, streamed Responses are a protocol error.

2. As an optimization, a Response can be stored into the CSR for the Client rather than allocating a separate CSR for a response buffer. However, if multiple responses are handled, the code must be careful to perform duplicate detection on the Response stored there as well as those queued. In addition, GetResponse must create a queued version of this Response before allowing it to be overwritten.

3. The handling of Group Responses has been omitted for brevity. Basically, a Response is accepted if there has been a Request received locally from the same Client and same Transaction that has not been responded to. In this case, the Response is delivered to the Server or queued.

4.8. Management Operations

VMTP uses management operations (invoked as remote procedure calls) to effectively acknowledge packet groups and request retransmissions. The following routine is invoked by the Client's management module on request from the Server.

```
NotifyVmtpClient( clientId,ctrl,receiveSeqNumber,transact,delivery,code)
    Get csr for clientId
    if none then return
    if RemoteClient( csr ) and not NotifyVmtpRemoteClient then
       return
|   else (for streaming)
|      Find csr with same LocalTransaction as transact
|      if csr is NULL then return
    if csr.State not AwaitingResponse then return
    if ctrl.PGcount then ack previous packet groups.
    select on code
      case OK:
        Notify ack'ed segment blocks from delivery
        Clear csr.RetransCount;
        Timeout( csr, TC1(csr.Server), LocalClientTimeout )
        return
      case RETRY:
        Set csr.TransmissionMask to missing segment blocks,
            as specified by delivery
        SendPacketGroup( csr )
        Timeout( csr, TC1(csr.Server), LocalClientTimeout )
      case RETRY_ALL
        Set csr.TransmissionMask to retransmit all blocks.
        SendPacketGroup( csr )
        Timeout( csr, TC1(csr.Server), LocalClientTimeout )
|       if streaming then
|          Restart transmission of packet groups,
|                starting from transact+1
         return
      case BUSY:
         if csr.TimeLimit exceeded then
             Set csr.Code to USER_TIMEOUT
             return Response to application
             return;
        Set csr.TransmissionMask for full retransmission
        Clear csr.RetransCount
        Timeout( csr, TC1(csr.Server), LocalClientTimeout )
        return
      case ENTITY_MIGRATED:
        Get new host address for entity
```

```
        Set csr.TransmissionMask for full retransmission
        Clear csr.RetransCount
        SendPacketGroup( csr )
        Timeout( csr, TC1(csr.Server), LocalClientTimeout )
        return

      case STREAMING_NOT_SUPPORTED:
        Record that server does not support streaming
        if CMG(csr) then forget this packet group
        else resend Request as separate packet group.
        return
      default:
         Set csr.Code to code
         return Response to application
         return;
    endselect
end NotifyVmtpClient
```

Notes:

1. The delivery parameter indicates the segment blocks received by the Server. That is, a 1 bit in the i-th position indicates that the i-th segment block in the segment data of the Request was received. All subsequent NotifyVmtpClient operations for this transaction should be set to acknowledge a superset of the segment blocks in this packet. In particular, the Client need not be prepared to retransmit the segment data once it has been acknowledged by a Notify operation.

4.8.1. HandleNoCSR

HandleNoCSR is called when a packet arrives for which there is no CSR matching the client field of the packet.

```
HandleNoCSR( p, psize )
    if Secure(p) then
        if SecureVMTP not supported then
            { Assume a Request }
            if not Multicast(p) then
                NotifyClient(NULL,p,SECURITY_NOT_SUPPORTED)
            return
        endif
        HandleRequestNoCSR( p, psize )
        return
    endif
```

```
    if p.Checksum not null then
        if not VerifyChecksum(p, psize) then return;
    if OppositeByteOrder(p) then ByteSwap( p, psize )
    if psize not equal sizeof(VmtpHeader) + 4*p.Length then
        NotifyClient(NULL, p, VMTP_ERROR )
        return

    if p.FuncCode is Response then
|       if Migrated( p.Client ) then
|           NotifyServer(csr, p ENTITY_MIGRATED )
|       else
            NotifyServer(csr, p, NONEXISTENT_ENTITY )
        return
    endif

    if p.FuncCode is Request then
       HandleRequestNoCSR( p, psize )
    return
end HandleNoCSR
```

Notes:

1. The node need only check to see if the client entity has migrated if in fact it supports migration of entities.

2. The procedure HandleRequestNoCSR is specified in Section 5.8.1. In the minimal client version, it need only handle Probe requests and can do so directly without allocating a new CSR.

4.9. Timeouts

A client with a message transaction in progress has a single timer corresponding to the first unacknowledged request message. (In the absence of streaming, this request is also the last request sent.) This timeout is handled as follows:

```
LocalClientTimeout( csr )
  select on csr.State
    case AwaitingResponse:
      if csr.RetransCount > MaxRetrans(csr.Server) then
             terminate Client's message transactions up to
             and including the current message transaction.
             set return code to KERNEL_TIMEOUT
          return
      increment csr.RetransCount
      Resend current packet group with APG set.
      Timeout( csr, TC2(csr.Server), LocalClientTimeout )
      return
    case ReceivingResponse:
      if DGMset(csr) or csr.RetransCount > Max then
         if MDMset(csr) then
            Set MCB.MsgDeliveryMask to blocks received.
         else
            Set csr.Code to BAD_REPLY_SEGMENT
         return to user Client
      endif
      increment csr.RetransCount
      NotifyServer with RETRY
      Timeout( csr, TC3(csr.Server), LocalClientTimeout )
      return
  end select
end LocalClientTimeout
```

Notes:

1. A Client can only request retransmission of a Response if the Response is not idempotent. If idempotent, it must retransmit the Request. The Server should generally support the MsgDeliveryMask for Requests that it treats as idempotent and that require multi-packet Responses. Otherwise, there is no selective retransmission for idempotent message transactions.

2. The current packet group is the last one transmitted. Thus, with streaming, there may be several packet groups outstanding that precede the current packet group.

3. The Request packet group should be retransmitted without the segment data, resulting in a single short packet in the retransmission. The Server must then send a NotifyVmtpClient with a RETRY or RETRY_ALL code to get the segment data transmitted as needed. This strategy minimizes the overhead on the network and the server(s) for retransmissions.

5. Server Protocol Operation

This section describes the operation of the server portion of the protocol in terms of the procedures for handling VMTP user events, packet reception events and timeout events. Each server is assumed to implement the client procedures described in the previous chapter. (This is not strictly necessary but it simplifies the exposition.)

5.1. Remote Client State Record Fields

The CSR for a server is extended with the following fields, in addition to the ones listed for the client version.

RemoteClient Identifier for remote client that sent the Request that this CSR is handling.

RemoteClientLink
Link to next CSR hashing to same hash index in the ClientMap.

RemoteTransaction
Transaction identifier for Request from remote client.

RemoteDelivery The segment blocks received so far as part of a Request or yet to be acknowledged as part of a Response.

VerifyInterval Time interval since there was confirmation that the remote Client was still valid.

RemotePrincipal Account identification, possibly including key and key timeout for secure communication.

5.2. Remote Client Protocol States

A CSR in the server end is in one of the following states.

AwaitingRequest Waiting for a Request packet group. It may be marked as waiting on a specific Client, or on any Client.

ReceivingRequest
Waiting to receive additional Request packets in a multi-packet group Request.

Responded The Response has been sent and the CSR is timing out, providing duplicate suppression and retransmission (if

the Response was not idempotent).

ResponseDiscarded
: Response has been acknowledged or has timed out so cannot be retransmitted. However, duplicates are still filtered and CSR can be reused for new message transaction.

Processing
: Executing on behalf of the Client.

Forwarded
: The message transaction has been forwarded to another Server that is to respond directly to the Client.

5.3. State Transition Diagrams

The CSR state transitions in the server are illustrated in Figure 5-1. The CSR generally starts in the AwaitingRequest state. On receipt of a Request, the Server either has an up-to-date CSR for the Client or else it sends a Probe request (as a separate VMTP message transaction) to the VMTP management module associated with the Client. In the latter case, the processing of the Request is delayed until a Response to the Probe request is received. At that time, the CSR information is brought up to date and the Request is processed. If the Request is a single-packet request, the CSR is then set in the Processing state to handle the request. Otherwise (a multi-packet Request), the CSR is put into the ReceivingResponse state, waiting to receive subsequent Request packets that constitute the Request message. It exits the ReceivingRequest state on timeout or on receiving the last Request packet. In the former case, the request is delivered with an indication of the portion received, using the MsgDelivery field if MDM is set. After request processing is complete, either the Response is sent and the CSR enters the Responded state or the message transaction is forwarded and the CSR enters the Forwarded state.

In the Responded state, if the Response is not marked as idempotent, the Response is retransmitted on receipt of a retransmission of the corresponding Request, on receipt of a NotifyVmtpServer operation requesting retransmission or on timeout at which time APG is set, requesting an acknowledgment from the Client. The Response is retransmitted some maximum number of times at which time the Response is discarded and the CSR is marked accordingly. If a Request or a NotifyVmtpServer operation is received expecting retransmission of the Response after the CSR has entered the ResponseDiscarded state, a NotifyVmtpClient operation is sent back (or invoked in the Client management module) indicating that the response was discarded unless the Request was multicast, in which case no action is taken. After a

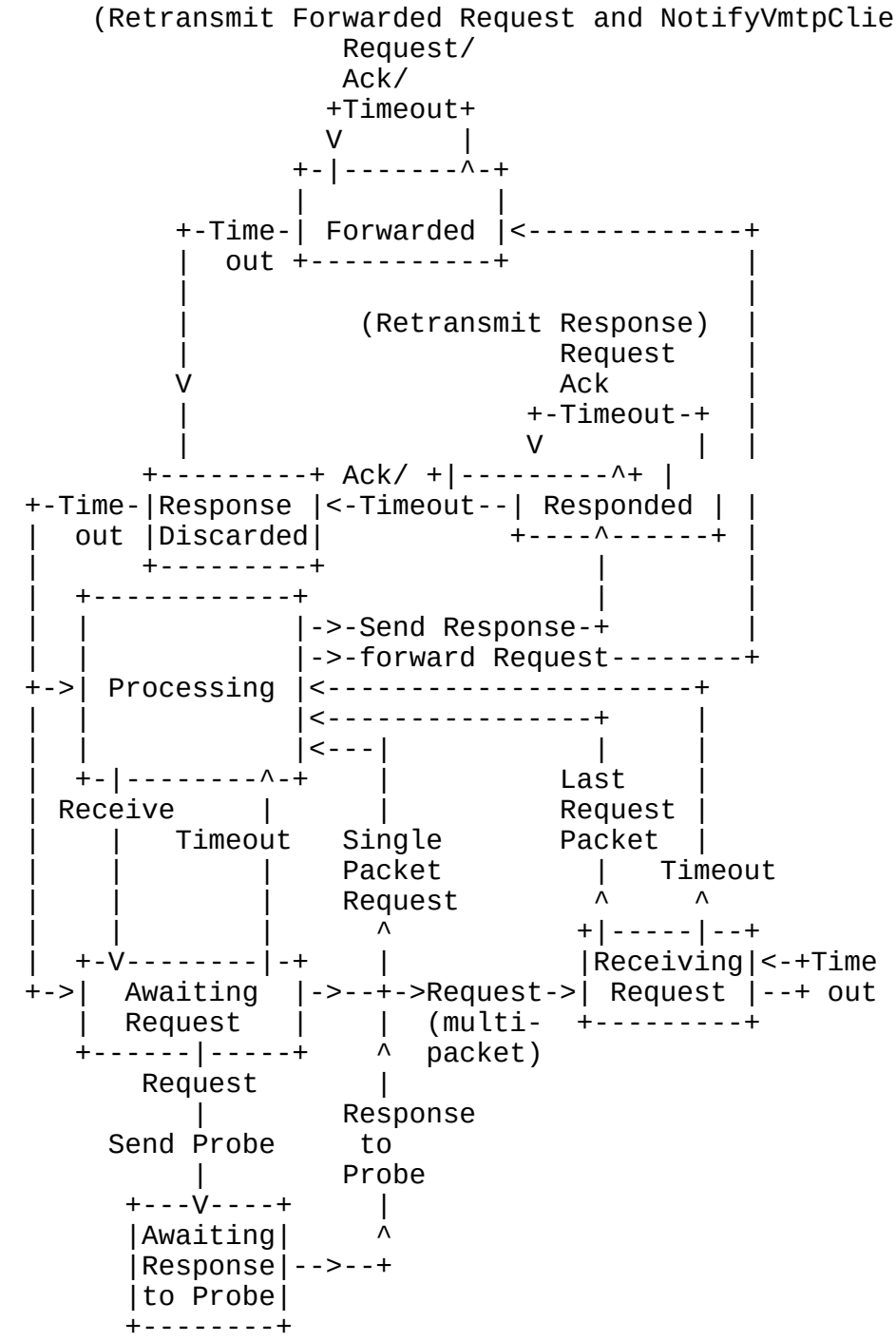


Figure 5-1: Remote Client State Transitions

timeout corresponding to the time required to filter out duplicates, the

CSR returns either to the AwaitingRequest state or to the Processing state. Note that "Ack" refers to acknowledgment by a Notify operation.

A Request that is forwarded leaves the CSR in the Forwarded state. In the Forwarded state, the forwarded Request is retransmitted periodically, expecting NotifyRemoteClient operations back from the Server to which the Request was forwarded, analogous to the Client behavior in the AwaitingResponse state. In this state, a NotifyRemoteClient from this Server acknowledges the Request or asks that it be retransmitted or reports an error. A retransmission of the Request from the Client causes a NotifyVmtpClient to be returned to the Client if APG is set. The CSR leaves the Forwarded state after timing out in the absence of NotifyRemoteClient operations from the forward Server or on receipt of a NotifyRemoteClient operation indicating the forward Server has sent a Response and received an acknowledgement. It then enters the ResponseDiscarded state.

Receipt of a new Request from the same Client aborts the current transaction, independent of its state, and initiates a new transaction unless the new Request is part of a run of message transactions. If it is part of a run of message transactions, the handling follows the state diagram except the new Request is not Processed until there has been a response sent to the previous transaction.

5.4. User Interface

The RPC or user interface to VMTP is implementation-dependent and may use systems calls, functions or some other mechanism. The list of requests that follow is intended to suggest the basic functionality that should be available.

AcceptMessage(reqmcb, segptr, segsize, client, transid, timeout)
: Accept a new Request message in the specified reqmcb area, placing the segment data, if any, in the area described by segptr and segsize. This returns the Server in the entityId field of the reqmcb and actual segment size in the segsize parameters. It also returns the Client and Transaction for this message transaction in the corresponding parameters. This procedure supports message semantics for request processing. When a server process executes this call, it blocks until a Request message has been queued for the server. AcceptMessage returns after the specified timeout period if a message has not been received by that time.

RespondMessage(responsemcb, client, transid, segptr)

Respond to the client with the specified response message and segment, again with message semantics.

RespondCall(responsemcb, segptr)
Respond to the client with the specified response message and segment, with remote procedure call semantics. This procedure does not return. The lightweight process that executes this procedure is matched to a stack, program counter, segment area and priority from the information provided in a ModifyService call, as specified in Appendix III.

ForwardMessage(requestmcb, transid, segptr, segsize, forwardserver)
Forward the client to the specified forwardserver with the request specified in mcb.

ForwardCall(requestmcb, segptr, segsize, forwardserver)
Forward the client transaction to the specified forwardserver with the request specified by requestmcb. This procedure does not return.

GetRemoteClientId()
Return the entityId for the remote client on whose behave the process is executing. This is only applicable in the procedure call model of request handling.

GetForwarder(client)
Return the entity that forwarded this Request, if any.

GetProcess(client)
Return an identifier for the process associated with this client entity-id.

GetPrincipal(client)
Return the principal associated with this client entity-id.

5.5. Event Processing

The following events may occur in VMTP servers.

- User Requests

    * Receive

        * Respond

        * Forward

        * GetForwarder

        * GetProcess

        * GetPrincipal

   - Packet Arrival

        * Request Packet

   - Management Operations

        * NotifyVmtpServer

   - Timeouts

        * Client State Record Timeout

The handling of these events is described in detail in the following subsections. The conventions of the previous chapter are followed, including the use of the various subroutines in the description.

5.6. Server User-invoked Events

A user event occurs when a VMTP server invokes one of the VMTP interface procedures.

5.6.1. Receive

```
AcceptMessage(reqmcb, segptr, segsize, client, transid, timeout)
    Locate server's request queue.
    if request is queued then
        Remember CSR associated with this Request.
        return Request in reqmcb, segptr and segsize
               and client and transaction id.
    Wait on server's request queue for next request
    up time timeout seconds.
end ReceiveCall
```

Notes:

1. If a multi-packet Request is partially received at the time of the AcceptMessage, the process waits until it completes.

2. The behavior of a process accepting a Request as a lightweight thread is similar except that the process executes using the Request data logically as part of the requesting Client process.

5.6.2. Respond

RespondCall is described as one case of the Respond transmission procedure; RespondMessage is similar.

```
RespondCall( responsemcb, responsesegptr )
    Locate csr for this client.
    Check segment data accessible, if any
    if local client then
        Handle locally
        return
    endif
    if responsemcb.Code is RESPONSE_DISCARDED then
        Mark as RESPONSE_DISCARDED
        return
    SendPacketGroup( csr )
    set csr.State to Responded.
    if DGM reply then { Idempotent }
        release segment data
        Timeout( csr, TS4(csr.Client), FreeCsr );
    else { Await acknowledgement or new Request else ask for ack. }
        Timeout( csr, TS5(csr.Client), RemoteClientTimeout )
end RespondCall
```

Notes:

1. RespondMessage is similar except the Server process must be synchronized with the release of the segment data (if any).

2. The non-idempotent Response with segment data is sent first without a request for an acknowledgement. The Response is retransmitted after time TS5(client) if no acknowledgment or new Request is received from the client in the meantime. At this point, the APG bit is sent.

3. The MCB of the Response is buffered in the client CSR, which remains for TS4 seconds, sufficient to filter old duplicates. The segment data (if any) must be retained intact until: (1)

after transmission if idempotent or (2) after acknowledged or timeout has occurred if not idempotent. Techniques such as copy-on-write might be used to keep a copy of the Response segment data without incurring the cost of a copy.

5.6.3. Forward

Forwarding is logically initiating a new message transaction between the Server (now acting as a Client) and the server to which the Request is forwarded. When the second server returns a Response, the same Response is immediately returned to the Client. The forwarding support in VMTP preserves these semantics while providing some performance optimizations in some cases.

```
ForwardCall( req, segptr, segsize, forwardserver )
    Locate csr for this client.
    Check segment data accessible, if any

    if local client or Request was multicast or secure
       or csr.ForwardCount == 15 then
        Handle as a new Send operation
        return
    if forwardserver is local then
        Handle locally
        return
    Set csr.funccode to Request
    Increment csr.ForwardCount
    Set csr.State to Responded
    SendPacketGroup( csr ) { To ForwardServer }
    Timeout( csr, TS4(csr.Client), FreeAlien )
end ForwardCall
```

Notes:

1. A Forward is logically a new call or message transaction. It must be really implemented as a new message transaction if the original Request was multicast or secure (with the optional further refinement that it can be used with a secure message transaction when the Server and ForwardServer are the same principal and the Request was not multicast).

2. A Forward operation is never handled as an idempotent operation because it requires knowledge that the ForwardServer will treat the forwarded operation as idempotent as well. Thus, a Forward operation that includes a segment should set APG on the first transmission of the

forwarded Request to get an acknowledgement for this data. Once the acknowledgement is received, the forwarding Server can discard the segment data, leaving only the basic CSR to handle retransmissions from the Client.

5.6.4. Other Functions

GetRemoteClient is a simple local query of the CSR. GetProcess and GetPrincipal also extract this information from the CSR. A server module may defer the Probe callback to the Client to get that information until it is requested by the Server (assuming it is not using secure communication and duplicate suppression is adequate without callback.) GetForwarder is implemented as a callback to the Client, using a GetRequestForwarder VMTP management operation. Additional management procedures for VMTP are described in Appendix III.

5.7. Request Packet Arrival

The basic packet reception follows that described for the Client routines. A Request packet is handled by the procedure HandleRequest.

```
HandleRequest( csr, p, psize )

    if LocalClient(csr) then
        { Forwarded Request on local Client }
        if csr.LocalTransaction != p.Transaction then return
        if csr.State != AwaitingResponse then return
        if p.ForwardCount < csr.ForwardCount then
           Discard Request and return.
        Find a CSR for Client as a remote Client.
        if not found then
            if packet group complete then
                handle as a local message transaction
                return
            Allocate and init CSR
            goto newTransaction
        { Otherwise part of current transaction }
        { Handle directly below. }n
    if csr.RemoteTransaction = p.Transaction then
      { Matches current transaction }
        if OldForward(p.ForwardCount,csr.ForwardCount) then
            return
        if p.ForwardCount > csr.ForwardCount then
          { New forwarded transaction }
            goto newTransaction
```

```
        { Otherwise part of current transaction }
        if csr.State = ReceivingRequest then
            if new segment data then retain in CSR segment area.
            if Request not complete then
               Timeout( csr, TS1(p.Client), RemoteClientTimeout )
               return;
            endif
            goto endPacketGroup
        endif
        if csr.State is Responded then
          { Duplicate }
            if csr.Code is RESPONSE_DISCARDED
               and Multicast(p) then
                return
            endif
            if not DGM(csr) then { Not idempotent }
                if SegmentData(csr) then set APG
                { Resend Response or Request, if Forwarded }
                SendPacketGroup( csr )
                timeout=if SegmentData(csr) then TS5(csr.Client)
                          else TS4(csr.Client)
                Timeout( csr, timeout, RemoteClientTimeout )
                return
            { Else idempotent - fall thru to newTransaction }
        else { Presume it is a retransmission }
            NotifyClient( csr, p, OK )
            return
   else if OldTransaction(csr.RemoteTransact,p.Transaction) then
        return
    { Otherwise, a new message transaction. }
newTransaction:
    Abort handling of previous transactions for this Client.

    if (NSRset(p) or NERset(p)) and NoStreaming then
        NotifyClient( csr, p, STREAMING_NOT_SUPPORTED )
        return
|   if NSRset(p) then { Streaming }
|     { Check that consecutive with previous packet group }
|       Find last packet group CSR from this client.
|      if p.Transaction not lastcsr.RemoteTransaction+1 mod 2**32
|         and not STIset(lastcsr) or
|        p.Transaction not lastcsr.RemoteTransaction+256 mod **32
|        then
|         { Out of order packet group }
|         NotifyClient(csr, p, BAD_TRANSACTION_ID )
|         return
|       endif
```

```
|       if lastcsr not completed then
|           NotifyClient( lastcsr, p, RETRY )
|       endif
|       if lastcsr available then use it for this packet group
|       else allocate and initialize new CSR
|       if CMG(lastcsr) then
|          Add segment data to lastcsr Request
|          Keep csr as record of this packet group.
|          Clear lastcsr.VerifyInterval
|      endif
|   else { First packet group }
        if MultipleRemoteClients(csr) then ScavengeCsrs(p.Client)
        Set csr.RemoteTransaction, csr.Priority
        Copy message and segment data to csr's segment area
         and set csr.PacketDelivery to that delivered.
        Clear csr.PacketDelivery
        Clear csr.VerifyInterval
        SaveNetworkAddress( csr, p )
    endif
    if packetgroup not complete then
        Timeout( csr, TS3(p.Client), RemoteClientTimeout )
        return;
    endif
endPacketGroup:
    { We have received complete packet group. }
    if APG(p) then NotifyClient( csr, p, OK )
    endif
|   if NERset(p) and CMG(p) then
|       Queue waiting for continuation packet group.
|       Timeout( csr, TS3(csr.Client), RemoteClientTimeout )
|       return
|   endif
    { Deliver request message. }
    if GroupId(csr.Server) then
        For each server identified by csr.Server
            Replicate csr and associated data segment.
            if CMDset(csr) and Server busy then
               Discard csr and data
            else
               Deliver or invoke csr for each Server.
            if not DGMset(csr) then queue for Response
            else Timeout( csr, TS4(csr.Client), FreeCsr )
        endfor
     else
       if CMDset(csr) and Server busy then
           Discard csr and data
        else
```

```
           Deliver or invoke csr for this server.
         if not DGMset(csr) then queue for Response
         else Timeout( csr, TS4(csr.Client), FreeCsr )
      endif
end HandleRequest
```

Notes:

1. A Request received that specifies a Client that is a local entity should be a Request forwarded by a remote server to a local Server.

2. An alternative structure for handling a Request sent to a group when there are multiple local group members is to create a remote CSR for each group member on reception of the first packet and deliver a copy of each packet to each such remote CSR as each packet arrives.

5.8. Management Operations

VMTP uses management operations (invoked as remote procedure calls) to effectively acknowledge packet groups and request retransmissions. The following routine is invoked by the Server's management module on request from the Client.

```
NotifyVmtpServer(server,clientId,transact,delivery,code)
    Find csr with same RemoteTransaction and RemoteClient
    as clientId and transact.
    if not found or csr.State not Responded then return
    if DGMset(csr) then
        if transmission of Response in progress then
            Abort transmission
            if code is migrated then
               restart transmission with new host addr.
        if Retry then Report protocol error
        return
    endif
    select on code
      case RETRY:
        if csr.RetransCount > MaxRetrans(clientId) then
             if response data segment then
                 Discard data and mark as RESPONSE_DISCARDED
|                if NERset(csr) and subsequent csr then
|                    Deallocate csr and use later csr for
|                    future duplicate suppression
|                endif
             return
        endif
        increment csr.RetransCount
        Set csr.TransmissionMask to missing segment blocks,
            as specified by delivery
        SendPacketGroup( csr )
        Timeout( csr, TS3(csr.Client), RemoteClientTimeout )
      case BUSY:
        if csr.TimeLimit exceeded then
            if response data segment then
                Discard data and mark as RESPONSE_DISCARDED
|               if NERset(csr) and subsequent csr then
|                   Deallocate csr and use later csr for
|                   future duplicate suppression
|               endif
             endif
        endif
        Set csr.TransmissionMask for full retransmission
        Clear csr.RetransCount
```

```
        Timeout( csr, TS3(csr.Server), RemoteClientTimeout )
        return

      case ENTITY_MIGRATED:
        Get new host address for entity
        Set csr.TransmissionMask for full retransmission
        Clear csr.RetransCount
        SendPacketGroup( csr )
        Timeout( csr, TS3(csr.Server), RemoteClientTimeout )
        return

      case default:
        Abort transmission of Response if in progress.
        if response data segment then
           Discard data and mark as RESPONSE_DISCARDED
           if NERset(csr) and subsequent csr then
               Deallocate csr and use later csr for
               future duplicate suppression
           endif
        return
    endselect
end NotifyVmtpServer
```

Notes:

1. A NotifyVmtpServer operation requesting retransmission of the Response is acceptable only if the Response was not idempotent. When the Response is idempotent, the Client must be prepared to retransmit the Request to effectively request retransmission of the Response.

2. A NotifyVmtpServer operation may be received while the Response is being transmitted. If an error return, as an efficiency, the transmission should be aborted, as suggested when the Response is a datagram.

3. A NotifyVmtpServer operation indicating OK or an error allows the Server to discard segment data and not provide for subsequent retransmission of the Response.

5.8.1. HandleRequestNoCSR

When a Request is received from a Client for which the node has no CSR, the node allocates and initializes a CSR for this Client and does a callback to the Client's VMTP management module to get the Principal, Process and other information associated with this Client. It also

checks that the TransactionId is correct in order to filter out duplicates.

```
HandleRequestNoCSR( p, psize )
|   if Secure(p) then
|       Allocate and init CSR
|       SaveSourceHostAddr( csr, p )
|       ProbeRemoteClient( csr, p, AUTH_PROBE )
|       if no response or error then
|          delete CSR
|          return
|       Decrypt( csr.Key, p, psize )
|        if p.Checksum not null then
|       if not VerifyChecksum(p, psize) then return;
|       if OppositeByteOrder(p) then ByteSwap( p, psize )
|       if psize not equal sizeof(VmtpHeader) + 4*p.Length then
|          NotifyClient(NULL, p, VMTP_ERROR )
|          return
|       HandleRequest( csr, p, psize )
|       return
    if Server does not exist then
        NotifyClient( csr, p, NONEXISTENT_ENTITY )
        return
    endif
    if security required by server then
        NotifyClient(csr, p, SECURITY_REQUIRED )
        return
    endif
    Allocate and init CSR
    SaveSourceHostAddr( csr, p );
    if server requires Authentication then
        ProbeRemoteClient( csr, p, AUTH_PROBE )
        if no response or error then
           delete CSR
           return
    endif
    { Setup immediately as a new message transaction }
    set csr.Server to p.Server
    set csr.RemoteTransaction to p.Transaction-1

    HandleRequest( csr, p, psize )
    endif
```

Notes:

1. A Probe request is always handled as a Request not requiring authentication so it never generates a callback Probe to the

Client.

2. If the Server host retains remote client CSR's for longer than the maximum packet lifetime and the Request retransmission time, and the host has been running for at least that long, then it is not necessary to do a Probe callback unless the Request is secure. A Probe callback can take place when the Server asks for the Process or PrincipalId associated with the Client.

5.9. Timeouts

The server must implement a timeout for remote client CSRs. There is a timeout for each CSR in the server.

```
RemoteClientTimeout( csr )
  select on csr.State
    case Responded:
        if RESPONSE_DISCARDED then
            mark as timed out
            Make a candidate for reuse.
            return
        if csr.RetransCount > MaxRetrans(Client) then
            discard Response
            mark CSR as RESPONSE_DISCARDED
            Timeout(csr, TS4(Client), RemoteClientTimeout)
            return
        increment csr.RetransCount
        { Retransmit Response or forwarded Request }
        Set APG to get acknowledgement.
        SendPacketGroup( csr )
        Timeout( csr, TS3(Client), RemoteClientTimeout )
        return
    case ReceivingRequest:
      if csr.RetransCount > MaxRetrans(csr.Client)
         or DGMset(csr) or NRTset(csr) then
          Modify csr.segmentSize and csr.MsgDelivery
          to indicate packets received.
          if MDMset(csr) then
              Invoke processing on Request
              return
          else
              discard Request and reuse CSR
              (Note: Need not remember Request discarded.)
              return
      increment csr.RetransCount
      NotifyClient( csr, p, RETRY )
      Timeout( csr, TS3(Client), RemoteClientTimeout )
      return
    default:
        Report error - invalid state for RemoteClientTimeout
    endselect
end RemoteClientTimeout
```

Notes:

1. When a CSR in the Responded state times out after discarding

the Response, it can be made available for reuse, either by the same Client or a different one. The CSR should be kept available for reuse by the Client for as long as possible to avoid unnecessary callback Probes.

6. Concluding Remarks

This document represents a description of the current state of the VMTP design. We are currently engaged in several experimental implementations to explore and refine all aspects of the protocol. Preliminary implementations are running in the UNIX 4.3BSD kernel and in the V kernel.

Several issues are still being discussed and explored with this protocol. First, the size of the checksum field and the algorithm to use for its calculation are undergoing some discussion. The author believes that the conventional 16-bit checksum used with TCP and IP is too weak for future high-speed networks, arguing for at least a 32-bit checksum. Unfortunately, there appears to be limited theory covering checksum algorithms that are suitable for calculation in software.

Implementation of the streaming facilities of VMTP is still in progress. This facility is expected to be important for wide-area, long delay communication.

I. Standard VMTP Response Codes

The following are the numeric values of the response codes used in VMTP.

0	OK
1	RETRY
2	RETRY_ALL
3	BUSY
4	NONEXISTENT_ENTITY
5	ENTITY_MIGRATED
6	NO_PERMISSION
7	NOT_AWAITING_MSG
8	VMTP_ERROR
9	MSGTRANS_OVERFLOW
10	BAD_TRANSACTION_ID
11	STREAMING_NOT_SUPPORTED
12	NO_RUN_RECORD
13	RETRANS_TIMEOUT
14	USER_TIMEOUT
15	RESPONSE_DISCARDED
16	SECURITY_NOT_SUPPORTED
17	BAD_REPLY_SEGMENT
18	SECURITY_REQUIRED
19	STREAMED_RESPONSE
20	TOO_MANY_RETRIES
21	NO_PRINCIPAL

22	NO_KEY
23	ENCRYPTION_NOT_SUPPORTED
24	NO_AUTHENTICATOR
25-63	Reserved for future VMTP assignment.

Other values of the codes are available for use by higher level protocols. Separate protocol documents will specify further standard values.

Applications are free to use values starting at 0x00800000 (hex) for application-specific return values.

II. VMTP RPC Presentation Protocol

For complete generality, the mapping of the procedures and the parameters onto VMTP messages should be defined by a RPC presentation protocol. In the absence of an accepted standard protocol, we define an RPC presentation protocol for VMTP as follows.

Each procedure is assigned an identifying Request Code. The Request code serves effectively the same as a tag field of variant record, identifying the format of the Request and associated Response as a variant of the possible message formats.

The format of the Request for a procedure is its Request Code followed by its parameters sequentially in the message control block until it is full.

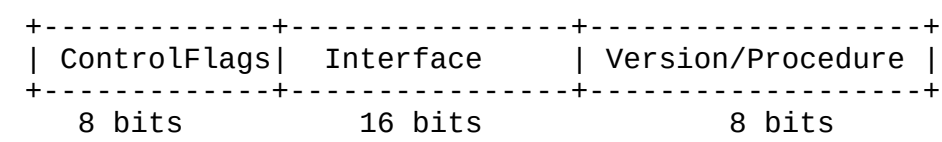
The remaining parameters are sent as part of the message segment data formatted according to the XDR protocol (RFC ??). In this case, the size of the segment is specified in the SegmentSize field.

The Response for a procedure consists of a ResponseCode field followed by the return parameters sequentially in the message control block, except if there is a parameter returned that must be transmitted as segment data, its size is specified in the SegmentSize field and the parameter is stored in the SegmentData field.

Attributes associated with procedure definitions should indicate the Flags to be used in the RequestCode. Request Codes are assigned as described below.

II.1. Request Code Management

Request codes are divided into Public Interface Codes and application-specific, according to whether the PIC value is set. An interface is a set of request codes representing one service or module function. A public interface is one that is to be used in multiple independently developed modules. In VMTP, public interface codes are allocated in units of 256 structured as



An interface is free to allocate the 8 bits for version and procedure as desired. For example, all 8 bits can be used for procedures. A module requiring more than 256 Version/Procedure values can be allocated

multiple Interface values. They need not be consecutive Interface values.

III. VMTP Management Procedures

Standard procedures are defined for VMTP management, including creation, deletion and query of entities and entity groups, probing to get information about entities, and updating message transaction information at the client or the server.

The procedures are implemented by the VMTP manager that constitutes a portion of every complete VMTP module. Each procedure is invoked by sending a Request to the VMTP manager that handles the entity specified in the operation or the local manager. The Request sent using the normal Send operation with the Server specified as the well-known entity group VMTP_MANGER_GROUP, using the CoResident Entity mechanism to direct the request to the specific manager that should handle the Request. (The ProbeEntity operation is multicast to the VMTP_MANAGER_GROUP if the host address for the entity is not known locally and the host address is determined as the host address of the responder. For all other operations, a ProbeEntity operation is used to determine the host address if it is not known.) Specifying co-resident entity 0 is interpreted as the co-resident with the invoking process. The co-resident entity identifier may also specify a group in which case, the Request is sent to all managers with members in this group.

The standard procedures with their RequestCode and parameters are listed below with their semantics. (The RequestCode range 0xVV000100 to 0xVV0001FF is reserved for use by the VMTP management routines, where VV is any choice of control flags with the PIC bit set. The flags are set below as required for each procedure.)

0x05000101 - ProbeEntity(CREntity, entityId, authDomain) -> (code, <staterec>)
Request and return information on the specified entity in the specified authDomain, sending the Request to the VMTP management module coresident with CREntity. An error return is given if the requested information cannot be provided in the specified authDomain. The <staterec> returned is structured as the following fields.

Transaction identifier
The current or next transaction identifier being used by the probed entity.

ProcessId: 64 bits
Identifier for client process. The meaning of this is specified as part of

the Domain definition.

PrincipalId The identifier for the principal or account associated with the process specified by ProcessId. The meaning of this field is specified as part of the Domain definition.

EffectivePrincipalId
The identifier for the principal or account associated with the Client port, which may be different from the PrincipalId especially if this is an nested call. The meaning of this field is specified as part of the Domain definition.

The code field indicates whether this is an error response or not. The codes and their interpretation are:

OK
No error. Probe was completed OK.

NONEXISTENT_ENTITY
Specified entity does not exist.

ENTITY_MIGRATED
The entity has migrated and is no longer at the host to which the request was sent.

NO_PERMISSION
Entity has refused to provide ProbeResponse.

VMTP_ERROR
The Request packet group was in error relative to the VMTP protocol specification.

"default"
Some type of error - discard ProbeResponse.

0x0D000102 - AuthProbeEntity(CREntity,entityId,authDomain,randomId) -> (code,ProbeAuthenticator,EncryptType,EntityAuthenticator)

Request authentication of the entity specified by entityId from the VMTP manager coresident with CREntity in authDomain authentication domain, returning the

information contained in the return parameters. The fields are set the same as that specified for the basic ProbeResponse except as noted below.

ProbeAuthenticator
: 20 bytes consisting of the EntityId, the randomId and the probed Entity's current Transaction value plus a 32-bit checksum for these two fields (checksummed using the standard packet Checksum algorithm), all encrypted with the Key supplied in the Authenticator.

EncryptType
: An identifier that identifies the variant of encryption method being used by the probed Entity for packets it transmits and packets it is able to receive. (See Appendix V.) The high-order 8 bits of the EncryptType contain the XOR of the 8 octets of the PrincipalId associated with private key used to encrypt the EntityAuthenticator. This value is used by the requestor or Client as an aid in locating the key to decrypt the authenticator.

EntityAuthenticator
: (returned as segment data) The ProcessId, PrincipalId, EffectivePrincipal associated with the ProbedEntity plus the private encryption/decryption key and its lifetime limit to be used for communication with the Entity. The authenticator is encrypted with a private key associated with the Client entity such that it can be neither read nor forged by a party not trusted by the Client Entity. The format of the Authenticator in the message segment is shown in detail in Figure III-1.

Key: 64 bits
: Encryption key to be used for encrypting and decrypting packets sent to and received from the probed Entity. This is the "working" key for packet transmissions. VMTP only uses private

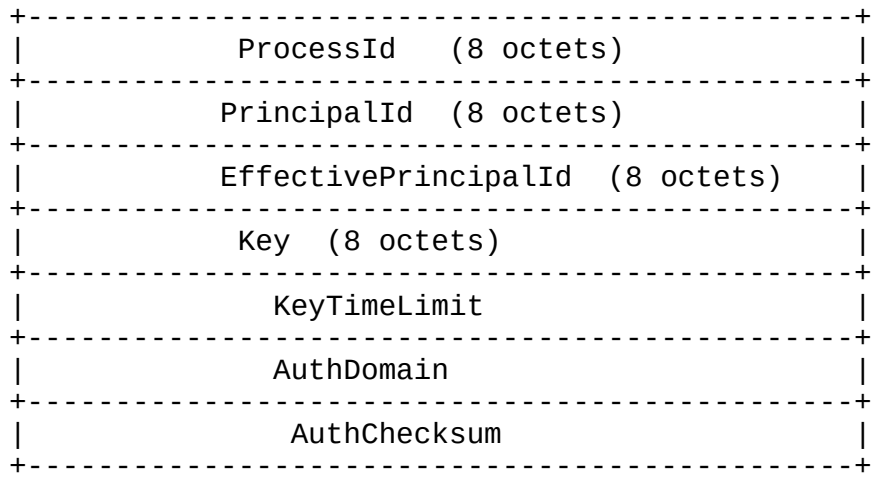


Figure III-1: Authenticator Format

key encryption for data transmission.

KeyTimeLimit: 32 bits
The time in seconds since Dec. 31st, 1969 GMT at which one should cease to use the Key.

AuthDomain: 32 bits
The authentication domain in which to interpret the principal identifiers. This may be different from the authDomain specified in the call if the Server cannot provide the authentication information in the request domain.

AuthChecksum: 32 bits
Contains the checksum (using the same Checksum algorithm as for packet) of KeyTimeLimit, Key, PrincipalId and EffectivePrincipalId.

Notes:

1. A authentication Probe Request and Response are sent unencrypted in general because it is used prior to there being a secure channel. Therefore, specific fields or groups of fields checksummed and encrypted to prevent unauthorized modification or forgery. In

particular, the ProbeAuthenticator is checksummed and encrypted with the Key.

2. The ProbeAuthenticator authenticates the Response as responding to the Request when its EntityId, randomId and Transaction values match those in the Probe request. The ProbeAutenticator is bound to the EntityAutenticator by being encrypted by the private Key contained in that authenticator.

3. The authenticator is encrypted such that it can be decrypted by a private key, known to the Client. This authenticator is presumably obtained from a key distribution center that the Client trusts. The AuthChecksum prevents undetected modifications to the authenticator.

0x05000103 - ProbeEntityBlock(entityId) -> (code, entityId)
Check whether the block of 256 entity identifiers associated with this entityId are in use. The entityId returned should match that being queried or else the return value should be ignored and the operation redone.

0x05000104 - QueryVMTPNode(entityId) -> (code, MTU, flags, authdomain, domains, authdomains, domainlist)
Query the VMTP management module for entityId to get various module- or node-wide parameters, including: (1) MTU - Maximum transmission unit or packet size handled by this node. (2) flags- zero or more of the following bit fields:

1	Handles streamed Requests.
2	Can issue streamed message transactions for clients.
4	Handles secure Requests.
8	Can issue secure message transactions.

The authdomain indicates the primary authentication domain supported. The domains and authdomains parameters indicate the number of entity domains and authentication domains supported by this node, which are listed in the data segment parameter domainlist if

either parameter is non-zero. (All the entity domains precede the authentication domains in the data segment.)

0x05000105 - GetRequestForwarder(CREntity, entityId1) -> (code, entityId2, principal, authDomain)
Return the forwarding server's entity identifer and principal for the forwarder of entityId1. CREntity should be zero to get the local VMTP management module.

0x05000106 - CreateEntity(entityId1) -> (code, entityId2)
Create a new entity and return its entity identifier in entityId2. The entity is created local to the entity specified in entityId1 and local to the requestor if entityId1 is 0.

0x05000107 - DeleteEntity(entityId) -> (code)
Delete the entity specified by entityId, which may be a group. If a group, the deletion is only on a best efforts basis. The client must take additional measures to ensure complete deletion if required.

0x0D000108 -QueryEntity(entityId) -> (code, descriptor)
Return a descriptor of entityId in arg of a maximum of segmentSize bytes.

0x05000109 - SignalEntity(entityId, arg)->(code)
Send the signal specified by arg to the entity specified by entityId. (arg is 32 bits.)

0x0500010A - CreateGroup(CREntity,entityGroupId,entityId,perms)->(code)
Request that the VMTP manager local to CREntity create an new entity group, using the specified entityGroupId with entityId as the first member and permissions "perms", a 32-bit field described later. The invoker is registered as a manager of the new group, giving it the permissions to add or remove members. (Normally CREntity is 0, indicating the VMTP manager local to the requestor.)

0x0500010B - AddToGroup(CREntity, entityGroupId, entityId, perms)->(code)
Request that the VMTP manager local to CREntity add the specified entityId to the entityGroupId with the specified permissions. If entityGroupId specifies a restricted group, the invoker must have permission to add members to the group, either because the invoker is

a manager of the group or because it was added to the group with the required permissions. If CREntity is 0, then the local VMTP manager checks permissions and forwards the request with CREntity set to entityId and the entityId field set to a digital signature (see below) of the Request by the VMTP manager, certifying that the Client has the permissions required by the Request. (If entityGroupId specifies an unrestricted group, the Request can be sent directly to the handling VMTP manager by setting CREntity to entityId.)

0x0500010C - RemoveFromGroup(CREntity, entityGroupId, entityId)->(code)
Request that the VMTP manager local to CREntity remove the specified entityId from the group specified by entityGroupId. Normally CREntity is 0, indicating the VMTP manager local to the requestor. If CREntity is 0, then the local VMTP manager checks permissions and forwards the request with CREntity set to entityId and the entityId field a digital signature of the Request by the VMTP manager, certifying that the Client has the permissions required by the Request.

0x0500010D - QueryGroup(entityId)->(code, record)...
Return information on the specified entity. The Response from each responding VMTP manager is (code, record). The format of the record is (memberCount, member1, member2, ...). The Responses are returned on a best efforts basis; there is no guarantee that responses from all managers with members in the specified group will be received.

0x0500010E - ModifyService(entityId,flags,count,pc,threadlist)->(code, count)
Modify the service associated with the entity specified by entityId. The flags may indicate a message service model, in which case the call "count" parameter indicates the maximum number of queued messages desired; the return "count" parameter indicates the number of queued message allowed. Alternatively, the "flags" parameters indicates the RPC thread service model, in which case "count" threads are requested, each with an inital program counter as specified and stack, priority and message receive area indicated by the threadlist. In particular, "threadlist" consists of "count" records of the form
(priority,stack,stacksize,segment,segmentsize), each one assigned to one of the threads. Flags defined for the

"flags" parameter are:

1 THREAD_SERVICE - otherwise the message model.

2 AUTHENTICATION_REQUIRED - Sent a Probe request to determine principal associated with the Client, if not known.

4 SECURITY_REQUIRED - Request must be encrypted or else reject.

8 INCREMENTAL - treat the count value as an increment (or decrement) relative to the current value rather than an absolute value for the maximum number of queued messages or threads.

In the thread model, the count must be a positive increment or else 0, which disables the service. Only a count of 0 terminates currently queued requests or in-progress request handling.

0x4500010F - NotifyVmtpClient(client,cntrl,recSeq,transact,delivery,code) ->()

Update the state associated with the transaction specified by client and transact, an entity identifier and transaction identifier, respectively. This operation is normally used only by another VMTP management module. (Note that it is a datagram operation.) The other parameters are as follows:

ctrl A 32-bit value corresponding to 4th 32-bit word of the VMTP header of a Response packet that would be sent in response to the Request that this is responding to. That is, the control flags, ForwardCount, RetransmitCount and Priority fields match those of the Request. (The NRS flag is set if the receiveSeqNumber field is used.) The PGCount subfield indicates the number of previous Request packet groups being acknowledged by this Notify operation. (The bit fields that are reserved in

this word in the header are also reserved here and must be zero.)

recSeq — Sequence number of reception at the Server if the NRS flag is set in the ctrl parameter, otherwise reserved and zero. (This is used for sender-based logging of message activity for replay in case of failure - an optional facility.)

delivery — Indicates the segment blocks of the packet group have been received at the Server.

code — indicates the action the client should take, as described below.

The VMTP management module should take action on this operation according to the code, as specified below.

OK — Do nothing at this time, continue waiting for the response with a reset timer.

RETRY — Retransmit the request packet group immediately with at least the segment blocks that the Server failed to receive, the complement of those indicated by the delivery parameter.

RETRY_ALL — Retransmit the request packet group immediately with at least the segment blocks that the Server failed to receive, as indicated by the delivery field plus all subsequently transmitted packets that are part of this packet run. (The latter is applicable only for streamed message transactions.)

BUSY — The server was unable to accept the Request at this time. Retry later if desired to continue with the message transaction.

NONEXISTENT_ENTITY — Specified Server entity does not exist.

ENTITY_MIGRATED The server entity has migrated and is no longer at the host to which the request was sent. The Server should attempt to determine the new host address of the Client using the VMTP management ProbeEntity operation (described earlier).

NO_PERMISSION Server has not authorized reception of messages from this client.

NOT_AWAITING_MSG
The conditional message delivery bit was set for the Request packet group and the Server was not waiting for it so the Request packet group was discarded.

VMTP_ERROR The Request packet group was in error relative to the VMTP protocol specification.

BAD_TRANSACTION_ID
Transaction identifier is old relative to the transaction identifier held for the Client by the Server.

STREAMING_NOT_SUPPORTED
Server does not support multiple outstanding message transactions from the same Client, i.e. streamed message transactions.

SECURITY_NOT_SUPPORTED
The Request was secure and this Server does not support security.

SECURITY_REQUIRED
The Server is refusing the Request because it was not encrypted.

NO_RUN_RECORD Server has no record of previous packets in this run of packet groups. This can occur if the first packet group is lost or if the current packet group is sent significantly later than the last one and the Server has discarded its client state record.

0x45000110 - NotifyVmtpServer(server,client,transact,delivery,code)->()

Update the server state associated with the transaction specified by client and transact, an entity identifier and transaction identifier, respectively. This operation is normally used only by another VMTP management module. (Note that it is a datagram operation.) The other parameters are as follows:

delivery — Indicates the segment blocks of the Response packet group that have been received at the Client.

code — indicates the action the Server should take, as listed below.

The VMTP management module should take action on this operation according to the code, as specified below.

OK — Client is satisfied with Response data. The Server can discard the response data, if any.

RETRY — Retransmit the Response packet group immediately with at least the segment blocks that the Client failed to receive, as indicated by the delivery parameter. (The delivery parameter indicates those segment blocks received by the Client).

RETRY_ALL — Retransmit the Response packet group immediately with at least the segment blocks that the Client failed to receive, as indicated by the (complement of) the delivery parameter. Also, retransmit all Response packet groups send subsequent to the specified packet group.

NONEXISTENT_ENTITY — Specified Client entity does not exist.

ENTITY_MIGRATED — The Client entity has migrated and is no longer at the host to which the response was sent.

RESPONSE_DISCARDED

The Response was discarded and no longer of interest to the Client. This may occur if the conditional message delivery bit was set for the Response packet group and the Client was not waiting for it so the Response packet group was discarded.

VMTP_ERROR The Response packet group was in error relative to the VMTP protocol specification.

0x41000111 -
NotifyRemoteVmtpClient(client,ctrl,recSeq,transact,delivery, code->()

The same as NotifyVmtpClient except the co-resident addressing is not used. This operation is used to update client state that is remote when a Request is forwarded.

Note the use of the CRE bit in the RequestCodes to route the request to the correct VMTP management module(s) to handle the request.

III.1. Entity Group Management

An entity in a group has a set of permissions associated with its membership, controling whether it can add or remove others, whether it can remove itself, and whether others can remove it from the group. The permissions for entity groups are as follows:

```
VMTP_GRP_MANAGER    0x00000001 { Manager of group. }
VMTP_REM_BY_SELF    0x00000002 { Can be removed self. }
VMTP_REM_BY_PRIN    0x00000004 { Can be rem'ed by same principal}
VMTP_REM_BY_OTHE    0x00000008 { Can be removed any others. }
VMTP_ADD_PRIN       0x00000010 { Can add by same principal. }
VMTP_ADD_OTHE       0x00000020 { Can add any others. }
VMTP_REM_PRIN       0x00000040 { Can remove same principal. }
VMTP_REM_OTHE       0x00000080 { Can remove any others. }
```

To remove an entity from a restricted group, the invoker must have permission to remove that entity and the entity must have permissions that allow it to be removed by that entity. With an unrestricted group, only the latter condition applies.

With a restricted group, a member can only be added by another entity with the permissions to add other entities. The creator of a group is given full permissions on a group. A entity adding another entity to a

group can only give the entity it adds a subset of its permissions. With unrestricted groups, any entity can add itself to the group. It can also add other entities to the group providing the entity is not marked as immune to such requests. (This is an implementation restriction that individual entities can impose.)

III.2. VMTP Management Digital Signatures

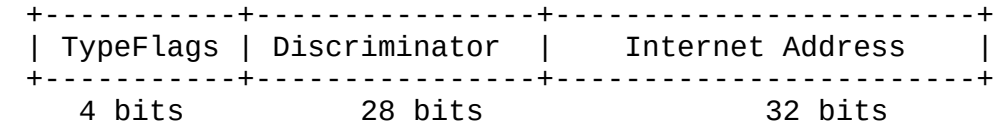
As mentioned above, the entityId field of the AddToGroup and RemoveFromGroup is used to transmit a digital signature indicating the permission for the operation has been checked by the sending kernel. The digital signature procedures have not yet been defined. This field should be set to 0 for now to indicate no signature after the CREntity parameter is set to the entity on which the operation is to be performed.

IV. VMTP Entity Identifier Domains

VMTP allows for several disjoint naming domains for its endpoints. The 64-bit entity identifier is only unique and meaningful within its domain. Each domain can define its own algorithm or mechanism for assignment of entity identifiers, although each domain mechanism must ensure uniqueness, stability of identifiers and host independence.

IV.1. Domain 1

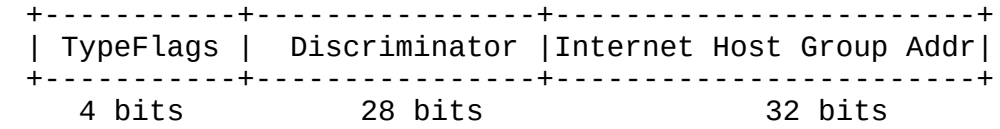
For initial use of VMTP, we define the domain with Domain identifier 1 as follows:



The Internet address is the Internet address of the host on which this entity-id is originally allocated. The Discriminator is an arbitrary value that is unique relative to this Internet host address. In addition, the host must guarantee that this identifier does not get reused for a long period of time after it becomes invalid. ("Invalid" means that no VMTP module considers in bound to an entity.) One technique is to use the lower order bits of a 1 second clock. The clock need not represent real-time but must never be set back after a crash. In a simple implementation, using the low order bits of a clock as the time stamp, the generation of unique identifiers is overall limited to no more than 1 per second on average. The type flags were described in Section 3.1.

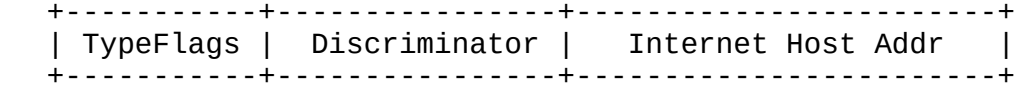
An entity may migrate between hosts. Thus, an implementation can heuristically use the embedded Internet address to locate an entity but should be prepared to maintain a cache of redirects for migrated entities, plus accept Notify operations indicating that migration has occurred.

Entity group identifiers in Domain 1 are structured in one of two forms, depending on whether they are well-known or dynamically allocated identifiers. A well-known entity identifier is structured as:



with the second high-order bit (GRP) set to 1. This form of entity identifier is mapped to the Internet host group address specified in the low-order 32 bits. The Discriminator distinguishes group identifiers using the same Internet host group. Well-known entity group identifiers should be allocated to correspond to the basic services provided by hosts that are members of the group, not specifically because that service is provided by VMTP. For example, the well-known entity group identifier for the domain name service should contain as its embedded Internet host group address the host group for Domain Name servers.

A dynamically allocated entity identifier is structured as:



with the second high-order bit (GRP) set to 1. The Internet address in the low-order 32 bits is a Internet address assigned to the host that dynamically allocates this entity group identifier. A dynamically allocated entity group identifier is mapped to Internet host group address 232.X.X.X where X.X.X are the low-order 24 bits of the Discriminator subfield of the entity group identifier.

We use the following notation for Domain 1 entity identifiers <10> and propose it use as a standard convention.

<flags>-<discriminator>-<Internet address>

where <flags> are [X]{BE,LE,RG,UG}[A]

X = reserved
BE = big-endian entity
LE = little-endian entity
RG = restricted group
UG = unrestricted group
A = alias

and <discriminator> is a decimal integer and <Internet address> is in standard dotted decimal IP address notation.

Examples:

<10> This notation was developed by Steve Deering.

BE-25593-36.8.0.49 is big-endian entity #25593 created on host 36.8.0.49.

RG-1-224.0.1.0 is the well-known restricted VMTP managers group.

UG-565338-36.8.0.77 is unrestricted entity group #565338 created on host 36.8.0.77.

LEA-7823-36.8.0.77 is a little-endian alias entity #7823 created on host 36.8.0.77.

This notation makes it easy to communicate and understand entity identifiers for Domain 1.

The well-known entity identifiers specified to date are:

VMTP_MANAGER_GROUP RG-1-224.0.1.0
Managers for VMTP operations.

VMTP_DEFAULT_BECLIENT BE-1-224.0.1.0
Client entity identifier to use when a (big-endian) host has not determined or been allocated any client entity identifiers.

VMTP_DEFAULT_LECLIENT LE-1-224.0.1.0
Client entity identifier to use when a (little-endian) host has not determined or been allocated any client entity identifiers.

Note that 224.0.1.0 is the host group address assigned to VMTP and to which all VMTP hosts belong.

Other well-known entity group identifiers will be specified in subsequent extensions to VMTP and in higher-level protocols that use VMTP.

IV.2. Domain 3

Domain 3 is reserved for embedded systems that are restricted to a single network and are independent of IP. Entity identifiers are allocated using the decentralized approach described below. The mapping of entity group identifiers is specific to the type of network being used and not defined here. In general, there should be a simple algorithmic mapping from entity group identifier to multicast address, similar to that described for Domain 1. Similarly, the values for default client identifier are specific to the type of network and not

defined here.

IV.3. Other Domains

Definition of additional VMTP domains is planned for the future. Requests for allocation of VMTP Domains should be addressed to the Internet protocol administrator.

IV.4. Decentralized Entity Identifier Allocation

The ProbeEntityBlock operation may be used to determine whether a block of entity identifiers is in use. ("In use" means valid or reserved by a host for allocation.) This mechanism is used to detect collisions in allocation of blocks of entity identifiers as part of the implementation of decentralized allocation of entity identifiers. (Decentralized allocation is used in local domain use of VMTP such as in embedded systems- see Domain 3.)

Basically, a group of hosts can form a Domain or sub-Domain, a group of hosts managing their own entity identifier space or subspace, respectively. As an example of a sub-Domain, a group of hosts in Domain 1 all identified with a particular host group address can manage the sub-Domain corresponding to all entity identifiers that contain that host group address. The ProbeEntityBlock operation is used to allocate the random bits of these identifiers as follows.

When a host requires a new block of entity identifiers, it selects a new block (randomly or by some choice algorithm) and then multicasts a ProbeEntityBlock request to the members of the (sub-)Domain some R times. If no response is received after R (re)transmissions, the host concludes that it is free to use this block of identifiers. Otherwise, it picks another block and tries again.

Notes:

1. A block of 256 identifiers is specified by an entity identifier with the low-order 8 bits all zero.

2. When a host allocates an initial block of entity identifiers (and therefore does not yet have a specified entity identifier to use) it uses VMTP_DEFAULT_BECLIENT (if big-endian, else VMTP_DEFAULT_LECLIENT if little-endian) as its client identifier in the ProbeEntityBlock Request and a transaction identifier of 0. As soon as it has allocated a block of entity identifiers, it should use these identifiers

for all subsequent communication. The default client identifier values are defined for each Domain.

3. The set of hosts using this decentralized allocation must not be subject to network partitioning. That is, the R transmissions must be sufficient to ensure that every host sees the ProbeEntityBlock request and (reliably) sends a response. (A host that detects a collision can retransmit the response multiple times until it sees a new ProbeEntityBlock operation from the same host/Client up to a maximum number of times.) For instance, a set of machines connected by a single local network may able to use this type of allocation.

4. To guarantee T-stability, a host must prevent reuse of a block of identifiers if any of the identifiers in the block are currently valid or have been valid less than T seconds previously. To this end, a host must remember recently used identifiers and object to their reuse in response to a ProbeEntityBlock operation.

5. Care is required in a VMTP implementation to ensure that Probe operations cannot be discarded due to lack of buffer space or queued or delayed so that a response is not generated quickly. This is required not only to detect collisions but also to provide accurate roundtrip estimates as part of ProbeEntity operations.

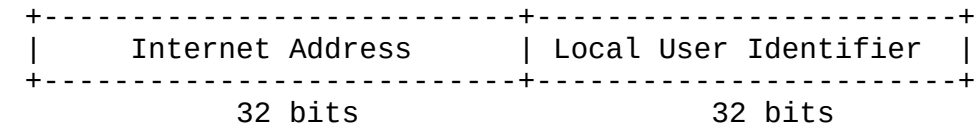
V. Authentication Domains

A VMTP authentication domain defines the format and interpretation for principal identifiers and encryption keys. In particular, an authentication domain must specify a means by which principal identifiers are allocated and guaranteed unique and stable. The currently defined authentication domains are as follows (0 is reserved).

Ideally, all entities within one entity domain are also associated with one authentication domain. However, authentication domains are orthogonal to entity domains. Entities within one domain may have different authentication domains. (In this case, it is generally necessary to have some correspondence between principals in the different domains.) Also, one entity identifier may be associated with multiple authentication domains. Finally, one authentication domain may be used across multiple entity domains.

V.1. Authentication Domain 1

A principal identifier is structured as follows.



The Internet Address may specify an individual host (such as a UNIX machine) or may specify a host group address corresponding to a cluster of machines operating under a single adminstration. In both cases, there is assumed to be an adminstration associated with the embedded Internet address that guarantees the uniqueness and stability of the User Identifier relative to the Internet address. In particular, that administration is the only one authorized to allocate principal identifiers with that Internet address prefix, and it may allocate any of these identifiers.

In authentication domain 1, the standard EncryptionQualifiers are:

0 Clear text - no encryption.

1 use 64-bit CBC DES for encryption and decryption.

V.2. Other Authentication Domains

Other authentication domains will be defined in the future as needed.

VI. IP Implementation

VMTP is designed to be implemented on the DoD IP Internet Datagram Protocol (although it may also be implemented as a local network protocol directly in "raw" network packets.)

VMTP is assigned the protocol number 81.

With a 20 octet IP header and one segment block, a VMTP packet is 600 octets. By convention, any host implementing VMTP implicitly agrees to accept VMTP/IP packets of at least 600 octets.

VMTP multicast facilities are designed to work with, and have been implemented using, the multicast extensions to the Internet [8] described in RFC 966 and 988. The wide-scale use of full VMTP/IP depends on the availability of IP multicast in this form.

VII. Implementation Notes

The performance and reliability of a protocol in operation is highly dependent on the quality of its implementation, in addition to the "intrinsic" quality of the protocol design. One of the design goals of the VMTP effort was to produce an efficiently implementable protocol. The following notes and suggestions are based on experience with implementing VMTP in the V distributed system and the UNIX 4.3 BSD kernel. The following is described for a client and server handling only one domain. A multi-domain client or server would replicate these structures for each domain, although buffer space may be shared.

VII.1. Mapping Data Structures

The ClientMap procedure is implemented using a hash table that maps to the Client State Record whether this entity is local or remote, as shown in Figure VII-1.

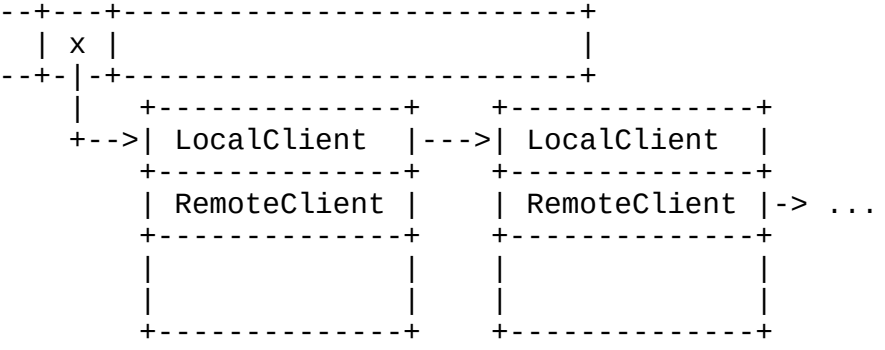


Figure VII-1: Mapping Client Identifier to CSR

Local clients are linked through the LocalClientLink, similarly for the RemoteClientLink. Once a CSR with the specified Entity Id is found, some field or flag indicates whether it is identifying a local or remote Entity. Hash collisions are handled with the overflow pointers LocalClientLink and RemoteClientLink (not shown) in the CSR for the LocalClient and RemoteClient fields, respectively. Note that a CSR representing an RPC request has both a local and remote entity identifier mapping to the same CSR.

The Server specified in a Request is mapped to a server descriptor using the ServerMap (with collisions handled by the overflow pointer.). The server descriptor is the root of a queue of CSR's for handling requests plus flags that modify the handling of the Request. Flags include:

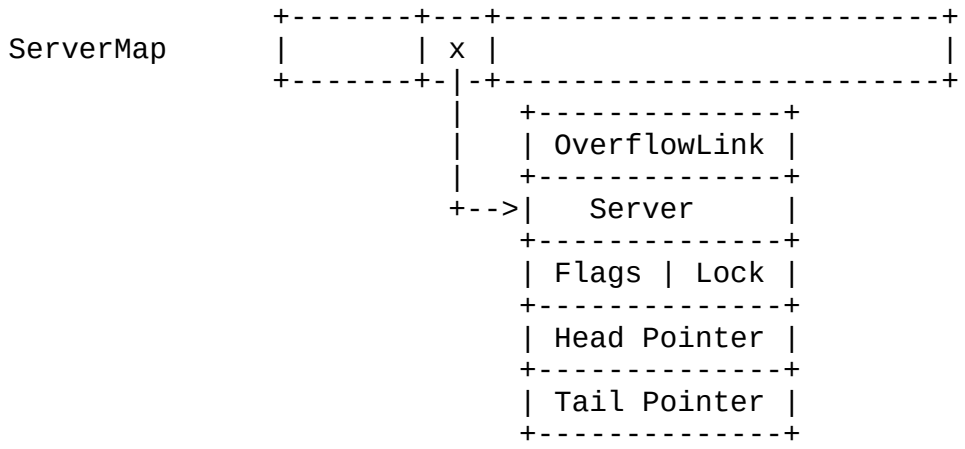


Figure VII-2: Mapping Server Identifiers

THREAD_QUEUE Request is to be invoked directly as a remote procedure invocation, rather than by a server process in the message model.

AUTHENTICATION_REQUIRED
Sent a Probe request to determine principal associated with the Client, if not known.

SECURITY_REQUIRED
Request must be encrypted or else reject.

REQUESTS_QUEUED Queue contains waiting requests, rather than free CSR's. Queue this request as well.

SERVER_WAITING The server is waiting and available to handle incoming Request immediately, as required by CMD.

Alternatively, the Server identifiers can be mapped to a CSR using the MapToClient mechanism with a pointer in the CSR refering to the server descriptor, if any. This scheme is attractive if there are client CSR's associated with a service to allow it to communicate as a client using VMTP with other services.

Finally, a similar structure is used to expand entity group identifiers to the local membership, as shown in Figure VII-3. A group identifier is hashed to an index in the GroupMap. The list of group descriptors rooted at that index in the GroupMap contains a group descriptor for each local member of the group. The flags are the group permissions defined in Appendix III.

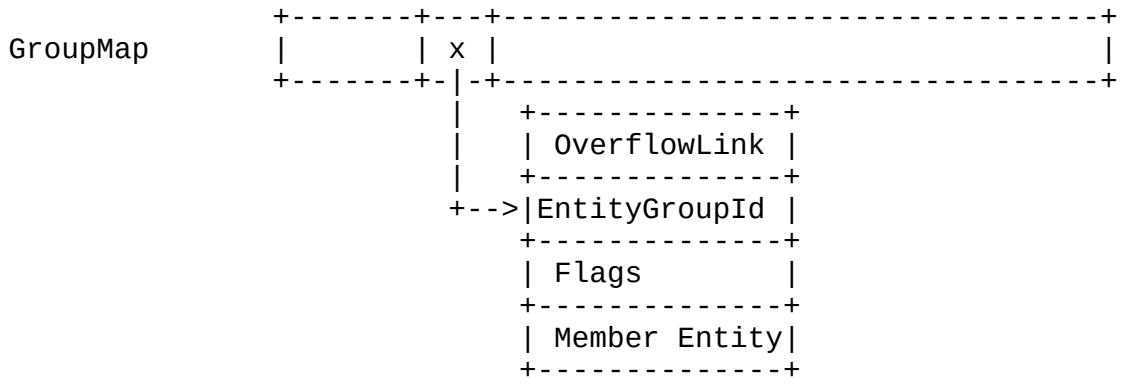


Figure VII-3: Mapping Group Identifiers

Note that the same pool of descriptors could be used for the server and group descriptors given that they are similar in size.

VII.2. Client Data Structures

Each client entity is represented as a client state record. The CSR contains a VMTP header as well as other bookkeeping fields, including timeout count, retransmission count, as described in Section 4.1. In addition, there is a timeout queue, transmission queue and reception queue. Finally, there is a ServerHost cache that maps from server entity-id records to host address, estimated round trip time, interpacket gap, MTU size and (optimally) estimated processing time for this server entity.

VII.3. Server Data Structures

The server maintains a heap of client state records (CSR), one for each (Client, Transaction). (If streams are not supported, there is, at worst, a CSR per Client with which the server has communicated with recently.) The CSR contains a VMTP header as well as various bookkeeping fields including timeout count, retransmission count. The server maintains a hash table mapping of Client to CSR as well as the transmission, timeout and reception queues. In a VMTP module implementing both the client and server functions, the same timeout queue and transmission queue are used for both.

VII.4. Packet Group transmission

The procedure SendPacketGroup(csr) transmits the packet group specified by the record CSR. It performs:

1. Fragmentation of the segment data, if any, into packets. (Note, segment data flagged by SDA bit.)

2. Modifies the VMTP header for each packet as required e.g. changing the delivery mask as appropriate.

3. Computes the VMTP checksum.

4. Encrypts the appropriate portion of the packet, if required.

5. Prepends and appends network-level header and trailer using network address from ServerHost cache, or from the responding CSR.

6. Transmits the packet with the interpacket gap specified in the cache. This may involve round-robin scheduling between hosts as well as delaying transmissions slightly.

7. Invokes the finish-up procedure specified by the CSR record, completing the processing. Generally, this finish-up procedure adds the record to the timeout queue with the appropriate timeout queue.

The CSR includes a 32-bit transmission mask that indicates the portions of the segment to transmit. The SendPacketGroup procedure is assumed to handle queuing at the network transmission queue, queuing in priority order according to the priority field specified in the CSR record. (This priority may be reflected in network transmission behavior for networks that support priority.)

The SendPacketGroup procedure only looks at the following fields of a CSR

- Transmission mask

- FuncCode

- SDA

- Client

- Server

- CoResidentEntity

- Key

It modifies the following fields

- Length

- Delivery

- Checksum

In the case of encrypted transmission, it encrypts the entire packet, not including the Client field and the following 32-bits.

If the packet group is a Response, (i.e. lower-order bit of function code is 1) the destination network address is determined from the Client, otherwise the Server. The HostAddr field is set either from the ServerHost cache (if a Request) or from the original Request if a Response, before SendPacketGroup is called.

The CSR includes a timeout and TTL fields indicating the maximum time to complete the processing and the time-to-live for the packets to be transmitted.

SendPacketGroup is viewed as the right functionality to implement for transmission in an "intelligent" network interface.

Finally, it appears preferable to be able to assume that all portions of the segment remain memory-resident (no page faults) during transmission. In a demand-paged systems, some form of locking is required to keep the segment data in memory.

VII.5. VMTP Management Module

The implementation should implement the management operations as a separate module that is invoked from within the VMTP module. When a Request is received, either from the local user level or the network, for the VMTP management module, the management module is invoked as a remote or local procedure call to handle this request and return a response (if not a datagram request). By registering as a local server, the management module should minimize the special-case code required for its invocation. The management module is basically a case statement that selects the operation based on the RequestCode and then invokes the specified management operation. The procedure implementing the management operation, especially operations like NotifyVmtpClient and

NotifyVmtpServer, are logically part of the VMTP module because they require full access to the basic data structures of the VMTP implementation.

The management module should be implemented so that it can respond quickly to all requests, particularly since the timing of management interactions is used to estimate round trip time. To date, all implementations of the management module have been done at the kernel level, along with VMTP proper.

VII.6. Timeout Handling

The timeout queue is a queue of CSR records, ordered by timeout count, as specified in the CSR record. On entry into the timeout queue, the CSR record has the timeout field set to the time (preferable in milliseconds or similar unit) to remain in the queue plus the finishup field set to the procedure to execute on removal on timeout from the queue. The timeout field for a CSR in the queue is the time relative to the record preceding it in the queue (if any) at which it is to be removed. Some system-specific mechanism decrements the time for the record at the front of the queue, invoking the finishup procedure when the count goes to zero.

Using this scheme, a special CSR is used to timeout and scan CSR's for non-recently pinged CSR's. That is, this CSR times out and invokes a finishup procedure that scans for non-recently pinged CSR that are "AwaitingResponse" and signals the request processing entity and deletes the CSR. It then returns to the timeout queue.

The timeout mechanism tends to be specific to an operating system. The scheme described may have to be adapted to the operating system in which VMTP is to be implemented.

This mechanism handles client request timeout and client response timeout. It is not intended to handle interpacket gaps given that these times are expected to be under 1 millisecond in general and possibly only a few microseconds.

VII.7. Timeout Values

Roundtrip timeout values are estimated by matching Responses or NotifyVmtpClient Requests to Request transmission, relying on the retransmitCount to identify the particular transmission of the Request that generated the response. A similar technique can be used with Responses and NotifyVmtpServer Requests. The retransmitCount is

incremented each time the Response is sent, whether the retransmission was caused by timeout or retransmission of the Request.

The ProbeEntity request is recommended as a basic way of getting up-to-date information about a Client as well as predictable host machine turnaround in processing a request. (VMTP assumes and requires an efficient, bounded response time implementation of the ProbeEntity operation.)

Using this mechanism for measuring RTT, it is recommended that the various estimation and smoothing techniques developed for TCP RTT estimation be adapted and used.

VII.8. Packet Reception

Logically a network packet containing a VMTP packet is 5 portions:

- network header, possibly including lower-level headers
- VMTP header
- data segment
- VMTP checksum
- network trailer, etc.

It may be advantageous to receive a packet fragmented into these portions, if supported by the network module. In this case, ideally the VMTP header may be received directly into a CSR, the data segment into a page that can be mapped, rather than copied, to its final destination, with VMTP checksum and network header in a separate area (used to extract the network address corresponding to the sender).

Packet reception is described in detail by the pseudo-code in Section 4.7.

With a response, normally the CSR has an associated segment area immediately available so delivery of segment data is immediate. Similarly, server entities should be "armed" with CSR’s with segment areas that provide for immediate delivery of requests. It is reasonable to discard segment data that cannot be immediately delivered in this way, providing that clients and servers are able to preallocate CSR’s with segment areas for requests and responses. In particular, a client should be able to provide some number of additional CSR’s for receiving multiple responses to a multicast request.

The CSR data structure is intended to be the interface data structure for an intelligent network interface. For reception, the interface is "armed" with CSR's that may point to segment areas in main memory, into which it can deliver a packet group. Ideally, the interface handles all the processing of all packets, interacting with the host after receiving a complete Request or Response packet group. An implementation should use an interface based on SendPacketGroup(CSR) and ReceivePacketGroup(CSR) to facilitate the introduction of an intelligent network interface.

ReceivePacketGroup(csr) provides the interface with a CSR descriptor and zero or more bytes of main memory to receive segment data. The CSR describes whether it is to receive responses (and if so, for which client) or requests (and if so for which server).

The procedure ReclaimCSR(CSR) reclaims the specified record from the interface before it has been returned after receiving the specified packet group.

A finishup procedure is set in the CSR to be invoked when the CSR is returned to the host by the normal processing sequence in the interface. Similarly, the timeout parameter is set to indicate the maximum time the host is providing for the routine to perform the specified function. The CSR and associated segment memory is returned to the host after the timeout period with an indication of progress after the timeout period. It is not returned earlier.

VII.9. Streaming

The implementation of streaming is optional in both VMTP clients and servers. Ideally, all performance-critical servers should implement streaming. In addition, clients that have high context switch overhead, network access overhead or expect to be communicating over long delay links should also implement streaming.

A client stream is implemented by allocating a CSR for each outstanding message transaction. A stream of transactions is handled similarly to multiple outstanding transactions from separate clients except for the interaction between consecutive numbered transactions in a stream.

For the server VMTP module, streamed message transactions to a server are queued (if accepted) subordinate to the first unprocessed CSR corresponding to this Client. Thus, streamed transactions from a given Client are always performed in the order specified by the transaction identifiers.

If a server does not implement streaming, it must refuse streamed message transactions using the NotifyVmtpClient operation. Also, all client VMTP's that support streaming must support the streamed interface to a server that does not support streaming. That is, it must perform the message transactions one at a time. Consequently, a program that uses the streaming interface to a non-streaming server experiences degraded performance, but not failure.

VII.10. Implementation Experience

The implementation experience to date includes a partial implementation (minus the streaming and full security) in the V kernel plus a similar preliminary implementation in the 4.3 BSD Unix kernel. In the V kernel implementation, the CSR's are part of the (lightweight) process descriptor.

The V kernel implementation is able to perform a VMTP message transaction with no data segment between two Sun-3/75's connected by 10 Mb Ethernet in 2.25 milliseconds. It is also able to transfer data at 4.7 megabits per second using 16 kilobyte Requests (but null checksums.) The UNIX kernel implementation running on Microvax II's achieves a basic message transaction time of 9 milliseconds and data rate of 1.9 megabits per second using 16 kilobyte Responses. This implementation is using the standard VMTP checksum.

We hope to report more extensive implementation experience in future revisions of this document.

VIII. UNIX 4.3 BSD Kernel Interface for VMTP

UNIX 4.3 BSD includes a socket-based design for program interfaces to a variety of protocol families and types of protocols (streams, datagrams). In this appendix, we sketch an extension to this design to support a transaction-style protocol. (Some familiarity with UNIX 4.2/3 IPC is assumed.) Several extensions are required to the system interface, rather than just adding a protocol, because no provision was made for supporting transaction protocols in the original design. These extensions include a new "transaction" type of socket plus new system calls invoke, getreply, probeentity, recreq, sendreply and forward.

A socket of type transaction bound to the VMTP protocol type IPPROTO_VMTP is created by the call

```
s = socket(AF_INET, SOCK_TRANSACT, VMTP);
```

This socket is bound to an entity identifier by

```
bind(s, &entityid, sizeof(entityid));
```

The first address/port bound to a socket is considered its primary name and is the one used on packet transmission. A message transaction is invoked between the socket named by s and the Server specified by mcb by

```
invoke(s, mcb, segptr, seglen, timeout );
```

The mcb is a message control block whose format was described in Section 2.4. The message control block specifies the request to send plus the destination Server. The response message control block returned by the server is stored in mcb when invoke returns. The invoking process is blocked until a response is received or the message transaction times out unless the request is a datagram request. (Non-blocking versions with signals on completion could also be provided, especially with a streaming implementation.)

For multicast message transactions (sent to an entity group), the next response to the current message transaction (if it arrives in less than timeout milliseconds) is returned by

```
getreply( s, mcb, segptr, maxseglen, timeout );
```

The invoke operation sent to an entity group completes as soon as the first response is received. A request is retransmitted until the first reply is received (assuming the request is not a datagram). Thus, the system does not retransmit while getreply is timing out even if no replies are available.

The state of an entity associated with entityId is probed using

```
probeentity( entityId, state );
```

A UNIX process acting as a VMTP server accepts a Request by the operation

```
recvreq(s, mcb, segptr, maxseglen );
```

The request message for the next queued transaction request is returned in mcb, plus the segment data of maximum length maxseglen, starting at segptr in the address space. On return, the message control block contains the values as set in invoke except: (1) the Client field indicates the Client that sent the received Request message. (2) the Code field indicates the type of request. (3) the MsgDelivery field indicates the portions of the segment actually received within the specified segment size, if MDM is 1 in the Code field. A segment block is marked as missing (i.e. the corresponding bit in the MsgDelivery field is 0) unless it is received in its entirety or it is all of the data in last segment contained in the segment.

To complete a transaction, the reply specified by mcb is sent to the client specified by the MCB using

```
sendreply(s, mcb, segptr );
```

The Client field of the MCB indicates the client to respond to.

Finally, a message transaction specified by mcb is forwarded to newserver as though it were sent there by its original invoker using

```
forward(s, mcb, segptr, timeout );
```

Index