An Architecture for Multi-Protocol Address Resolution

Willibald Doeringer and Mehdi Nassehi

IBM Research Division, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland
June 13, 1995 — e-mail: wdo@zurich.ibm.com, fax: +41-1-710 3608

Abstract: One of the essential prerequisites for truly distributed computing in heterogeneous networks, such as enterprise networks, is the independence of applications from communication protocols. A general architecture aiming at such an independence must allow applications to identify each other by addresses that may not be understood by the underlying networks. Therefore, it becomes necessary to map application addresses to network addresses. A versatile architecture for this address-mapping requirement has been developed by IBM and published as a Preliminary Specification by the X/Open consortium. The key features of this architecture are explained here and an overview of an early implementation is presented.

Keywords: Enterprise networks, heterogeneous networks, multi-protocol transport networking, address resolution, dynamic databases.

1. Introduction

The *Multiprotocol Transport Networking (MPTN)* architecture adopted by the X/Open consortium[1] [16, 17, 18] is a general solution for isolating distributed applications from communication protocols by enabling their use on both *native* and *non-native networks*, that is, those for which the applications were originally written as well as the others (Fig. 1). This solution represents a major evolutionary step beyond prior approaches [1, 2, 12] because it provides a general schema for specifying the semantics of transport services as well as a very flexible address structure to identify cooperating applications. Based on these conventions, the architecture defines how the respective semantics are supported over any transport protocol and specifies the procedure to map — where required — application addresses to those understood by the underlying network. The address-mapping protocol of the MPTN architecture is the focus of this article. The other features have been presented in [5].

To observe the need for *address mapping*, or *address resolution*, consider the situation depicted in Fig. 2 where a client and a server application designed to communicate over APPN/APPC, IBM's Advanced Peer-to-Peer Network, use a TCP/IP network as their transport mechanism. In order to make its services accessible, the server node first registers its APPN address LU-SERVER with the MPTN dynamic address registry[2], i.e., the *Address Mapper Server*, thus providing it with the coupling (LU-SERVER, IP.SERVER) 1. It receives a positive response for this registeration. When the APPC client requests a connection to its APPC server 2, its MPTN component intercepts the destination address, LU-SERVER, queries the Address Mapper Server, and receives IP.SERVER as the nodal IP address 3. Using the well-known port MPTN_PORT, the client node MPTN component establishes a TCP/IP connection with its peer at (IP.SERVER, MPTN_PORT) 4. It then negotiates the establishment of an APPN connection to LU-SERVER 5, and completes the initial connection request 6.

In the next section, the MPTN Address Mapper architecture [17] is presented. Section 3 describes an implementation of the architecture. The performance of this implementation is evaluated in Section 4. Section 5 is a summary of the paper.

2. The Architecture

Address mapping may be viewed as a set of operations on a database that maps byte strings of potentially different formats to each other. The requirements for heterogeneous networks are

- efficient support of an open-ended range of address formats,
- provision for dynamic changes of the contents of the mapping tables,

[1] X/Open Company Ltd., UK (XoSpecs@xopen.co.uk).

[2] MPTN nodes have prior knowledge of respective addresses of Address Mapper Servers.

- sufficiently fast operations to avoid disturbing the existing protocols by causing timeouts or early retries of operations,
- support of group-address and wildcard operations, and
- efficient scaling over a wide range of network sizes and numbers of database clients — from small LANs with tens of users to corporate networks with hundreds of thousands of users.

Address mapping is a well-known feature in many networking environments, and there are various solutions. However, none of them meet all of the above requirements. More specifically, let us consider the four typical approaches. The learning algorithms of (split) bridges [13] do not support multiple address formats, and their mapping tables are not dynamic. Algorithmic solutions [11] impose restrictions on the address formats in that any address space can only be mapped to a larger one. Configured mapping tables [4, 9, 10] preclude dynamic changes in the database. The distributed databases such as those found in LAN environments offer good performance but cannot support large networks [8]. Those such as *X.500*, on the other hand, have a hierarchical architecture designed specifically for large networks, but a search may need to go through many levels of the hierarchy, hence inflicting potentially long response times [15].

From its very beginning, it has been the objective of the MPTN effort to accommodate all of the above requirements. The principal design decisions made to meet this objective are discussed in the following presentation of the Address Mapper architecture.

2.1 Address Format

The representation of addressing information had to guarantee complete flexibility, avoid conflicts between the address spaces of different network protocols, and still be useful as a key into the mapping tables, i.e., the address database. The solution selected represents MPTN addresses as tuples *(AddressFamily, AddressMode, HostAddress, LocalAddress)*, where *AddressFamily* identifies the particular network type such as IP, APPN, OSI, or NetBIOS, *AddressMode* denotes particulars of the address such as individual or group addresses, *HostAddress* is an arbitrary byte string that indicates a network address such as a full IP or an OSI NSAP address, and finally the optional *LocalAddress* which represents another arbitrary byte string that identifies a specific 'port', i.e., T-selector (cf. Fig. 3).

2.2 Protocol

Referring back to Fig. 2 and recalling database fundamentals, one may infer that state-of-the-art address mapping services should at least support operations that add entries to the database and find entries based on keys and delete entries, where the keys are addresses of the above general structure. Reflecting the emphasis on simplicity in the MPTN design, only these three operations are provided. Let *UserAddress* denote the address of an application, i.e., a user of the transport services (cf. LU-SERVER), and *ProviderAddress* a real network address (cf. IP.SERVER). Address Mapper clients may issue the following requests to an Address Mapper Server and receive the pertinent positive or negative responses:

Register(*UserAddress, ProviderAddress, UserData***)** :
Indicates to the Address Mapper Server that *UserAddress* may be reached via the nodal network address *ProviderAddress*, and requests that the *UserData* be stored and returned together with the *ProviderAddress* upon subsequent retrieval of *UserAddress*. The request is acknowledged in a positive response or rejected with an appropriate error code. For example, registration of an already existing individual *UserAddress* by another client node is rejected with an error code of MPTN_ADDRESS_CONFLICT. The *Register* request corresponds to ABM_AM_REGISTER in [17].

Locate(*Type, Address***)** - *Type* specifies whether the second argument is a *UserAddress* or a *ProviderAddress*. If it is a *UserAddress*, the function retrieves all information stored at the Address Mapper Server about *UserAddress*. When found, the result is returned in a response carrying the pertinent *UserData*, if any, and the list of all *ProviderAddresses* associated with the *UserAddress* via prior Register requests. The receiving client may select one of the returned addresses to initiate communication, and may in addition cache (parts of) the retrieved information. If *Type* indicates that *Address* is a *ProviderAddress*, the function retrieves all the *UserAddresses* associated with the *ProviderAddress*. This type of *Locate* function was introduced to support network management and (automatic) error recovery for situations in which a particular *ProviderAddress* becomes unreachable, e.g., by switching to a possible alternate address. The *Locate* request corresponds to ABM_AM_LOCATE in [17].

DeRegister(*UserAddress, ProviderAddress***)** - Removes the entry from the database that relates the specified addresses. This makes the *UserAddress* unreachable via the *ProviderAddress* for any subsequently initiated communication. To reduce network traffic and support network management operations, both addresses may be specified as *Any*, such that whole sets of address pairs may be deleted — including the case of DeRegister(*Any*, *Any*), where all entries for the sending client are deleted from the address database. The *DeRegister* request corresponds to ABM_AM_DEREGISTER in [17].

The scalability of the architecture up to large enterprise networks with many thousands of client nodes implies the use of connectionless transport service for the transmission of the Address Mapper requests and responses. Otherwise, excessive amounts of storage for connection resources would be required at the server side, i.e., the Address Mapper Server. However, this decision necessitated a reliable protocol at the request-response level, which was designed on the basis of periodic retransmissions of unacknowledged requests up to a configured threshold. Whenever this threshold is reached, the MPTN component in the client node assumes that communication with the server has been lost and initiates remedial actions. This implicit liveness checking of the server is by nature asymmetric because typically only the clients will start sending unsolicited MPTN flows. However, for reasons of scalability and accuracy of the stored information, it is desirable that the address database be cleared of entries owned by clients that are no longer active — even in cases of ungraceful terminations. Hence, it was deemed necessary to have the server check the liveness of its clients periodically. But even with a reasonably long period of 30 seconds between probes, one could easily calculate that in large networks a server might consume a bandwidth in the order of one megabit per second just for this purpose — a result not well aligned with the requirement of scalability. The solution was to make the liveness checks against clients event-triggered. More specifically, if a client obtains a provider address from the server and uses it later to communicate to a node but cannot get a response, it informs the server that the provider address could not be reached. The server then checks the liveness of the node that owns the provider address by issuing a probe. The following two requests are used in this protocol.

NotFound*(UserAddress, ProviderAddress)* - The sending client attempted to reach *UserAddress* via *ProviderAddress* but failed. This request corresponds to ABM_AM_NOT_FOUND in [17].

Probe - The only request sent by an Address Mapper Server. It is transmitted using the reliable protocol described above. If no response is received, the respective client is assumed to be inactive and its entries in the address database are garbage collected or at least marked as 'unreliable', depending upon configuration parameters. This request corresponds to ABM_MA_R_U_THERE in [17].

2.3 Encoding of Line Flows

The encoding of MPTN line flows has been oriented along insights into protocol processing at high speeds as reported in recent years (cf. [3, 6, 14]): alignment along word or double-word boundaries, one-pass parsing, many common building blocks, frequently used formats devoid of optional fields. In addition, features have been integrated to ease the migration to enhanced versions of the protocols, such as flagging (parts of) optional fields which may safely be ignored by backlevel nodes. The overview of a sample implementation on page 5 demonstrates that it is indeed possible to implement a package of moderate code size that parses and forms MPTN line flows very efficiently. A typical example of the most common formats is shown in Fig. 4.

2.4 Address Database

The MPTN architecture does not prescribe how an Address Mapper Server internally organizes the information received from its clients. In other words, the implementation of the address database may be adapted to the operating environment. In small networks, such as LANs, RAM-based lists may be sufficient. In medium-sized networks, one may use an Address Mapper Server with non-volatile storage or a "hot" standby for transparent recovery of single failures, as shown in Fig. 5(a). In very large networks, one may deploy multiple Address Mapper Servers concurrently as frontends to a common, distributed database, as shown in Fig. 5(b). Respective clients will then be supplied with information about alternative servers to be used, and select the 'nearest' active one.

3. Implementation

We have designed, coded, tested, and evaluated a full implementation of the MPTN Address Mapper architecture. The code is organized into a *Client Library* and a *Server Library*, which are reentrant except for short sections of code that need to be serialized to protect the consistency of the address database.

3.1 Programming Interface

The nodal component that controls the MPTN-related activities, the *Common MPTN Manager (CMM)* [5, 16], invokes these libraries via well-defined interfaces shown in Fig. 6. Perfect portability across platforms[3] was achieved by coding in a suitable subset of ANSI C and delegating all system-dependent activities — such as the scheduling of processes or threads, supplying of a periodic timer tick, or sending and receiving of datagrams — to the controlling entity, i.e., the CMM.

A typical sequence of interactions across the programming interfaces with the resulting MPTN line flows is depicted in Fig. 7. The scenario is as follows:

1. The client CMM invokes the *Register* downcall function in the Client Library to register an address pair at the server. This interaction is most likely triggered by some application.

2. The *Register* function creates an MPTN line flow for a Register request and, by invoking a *SendDatagram* upcall, requests the client CMM to forward it to the server.

3. The client CMM transmits the Register datagram to the server node.

4. The client CMM returns control to the Client Library, while the server CMM receives the Register datagram and forwards it to the Server Library by invoking the *RecvdDatagram* downcall.

5. At the client side, the initial *Register* downcall ends with an MPTN_PENDING return code because the operation is only complete after a response is received from the server.

[3]At the time of writing, the code is being used for Windows, OS/2, AIX and OS/400.

6. The Server Library adds the address pair to its database and creates an MPTN line flow for a Register response, which it passes to its CMM for transmission.
7. The server CMM sends a datagram containing the Register response to the client node.
8. The server CMM returns execution to the Server Library with a good return code.
9. At the server node, the *RecvdDatagram* downcall ends successfully, while the client CMM forwards the received message to the Client Library via its *RecvdDatagram* downcall.
10. The Client Library accepts and processes the response datagram and invokes a *RegisterRsp* upcall.
11. The client CMM returns control to the Client Library with a good return code for the *RegisterRsp* upcall.
12. The Client Library returns control to the client CMM with a good return code for the *RecvdDatagram* downcall.

3.2 Modules

The module diagram shown in Fig. 8 reveals that Client and Server Libraries are built symmetrically, and that substantial reuse of code has thus been achieved.

The *Server Control* module controls the flow of execution through the Server Library. The overall design follows the so-called *assembly-line principle* in that the *Server Protocol, Format* and *Retransmission* modules as well as CMM upcall function are invoked only by the *Server Control* module and in a well-defined sequence. The *Control* module also manages the shared data structures, such as one communication record for each client containing, among other things, its status and its latest network address. It also contains the downcall functions of the Server Library, which are invoked by the server CMM.

The *Client Control* module controls the sequence of execution through the Client Library much like the Server Control module does through the Server Library. The same design principles were applied with slightly increased complexity because the module services address mapping requests generated at the local node as well as commands from the server. The externally accessible functions of the Client Library — the downcalls — are implemented in this module, and all upcalls against the local CMM are invoked from here.

The *Server Protocol* module processes the Address Mapper commands received from client nodes. That is, it interprets the received commands semantically using an internal representation of Address Mapper line formats, performs the required actions against the address database, and assembles appropriate responses to be transmitted.

The *Client Protocol* module processes the internally generated requests for address mapping services and Address Mapper commands received from the Address Mapper

Server again at the semantic level using an internal representation of MPTN line formats as syntactical representation. The module implements a typical client function in that in most cases it will transmit commands that have been triggered by an internal address mapping requests, and to which the server responds. This module is the counterpart of the Protocol Module in the corresponding Address Mapper Server.

The *Database* module maintains all registered information that relates user and provider addresses, which may be individual or group addresses. User addresses may be combined with bit masks to create wildcard entries in the database. The module implements a high-level interface that matches closely the various options for the add, find and delete operations specified by the architecture for the Register, Locate, DeRegister requests, respectively. This module is implemented based on the *Dynamic Prefix Trie* [7], a novel data structure with algorithms for building and maintaining a dynamic database of binary keys of arbitrary length. As a particular consequence of architectural requirements of the address-mapping protocol, i.e., the two types of Locate, the relation between user and provider addresses had to be implemented symmetrically. Also, the capabilities of handling group addresses and wildcard entries added considerable complexity to this module.

The *Retransmission* module provides a reliable transmission protocol at the level of server and client, because both rely only on the connectionless provider services. More specifically, this module provides functions to store and retrieve transmission buffers for retransmissions, and for identifying acknowledged buffers and those that are not acknowledged within a given time-out period.

The *Format* module performs the mapping between MPTN lineflows and their high-level internal syntactical representation. Thus, bit- and byte-level details are hidden from the invoking modules and the effect of changes of the layout of the MPTN line flows — which occurred frequently during the early design and implementation work — are completely localized.

4. Performance

The performance of the Address Mapper is evaluated here in terms of the average time it takes to perform *Locate*, *Register* and *DeRegister* operations. The measurements are obtained using a product-level implementation of the Address Mapper in ANSI C. The performance results are in terms of the CPU time on an IBM RISC System/6000 model 970, which has a SPEC'92 integer performance of 47.8. The code was compiled using the *xlc* compiler at the optimization level 2.

Table 1 shows the average request-processing times at the Address Mapper Server. These values are for a scenario in which there are 1000 nodes, each owning 50 user addresses and three provider addresses. Every node registers all its user addresses to be associated with all its provider addresses. All user and provider addresses as well as node addresses have a 4-byte *HostAddress*, derived from uniformly distributed random

Request	Server Control (msec.)	Server Protocol (msec.)	Database (msec.)	Format (msec.)	Total (msec.)
Locate	0.46	0.42	0.06	0.37	1.27
Register	0.41	0.12	4.97	0.37	5.89
DeRegister	0.35	0.12	1.21	0.29	1.97

Table 1: Average request-processing times at the Address Mapper Server.

integers, and a null *LocalAddress*. Every *Locate* is assumed to be for a user address that is already in the database, and every *Register* is assumed to be for a new address. The processing times associated with the retransmission module are included in that of the control module because they are not large enough to be of interest.

The variations among the processing times of the three requests are much larger in the protocol and database modules than those in the control and format modules. The protocol-handling time of *Locate* is larger than those of *Register* and *DeRegister* because the list of located provider addresses must be pruned of duplicates that may occur in the case of group addresses due to our specific implementation of the database module. On the other hand, the database processing times of *Register* and *DeRegister* are significantly longer than that of *Locate* because nodes must be added to and deleted from the search trees.

Based on the values shown in Table 1, we observe that the Address Mapper meets the performance requirements expected of a typical transaction server. The most important measure here is the total processing time of a Locate request, which is very close to 1 msec. The Register and DeRegister requests are expected to be issued much less frequently and, when issued, are not as time-critical as the Locate request. This is fortunate because in any database implementation, Registers and DeRegisters take longer to be processed.

We also measured the sensitivity of Locate processing times to the three parameters of our scenario, i.e., number of nodes, number of user addresses per node, and number of provider addresses per node. Let these parameters be denoted by N, U and P, respectively. The results show that the average Locate time increases logarithmically as N or U is increased; more specifically, for every factor of two increase in N or U, the average Locate time increased by about 2.5 μsec. The reason for this behavior is that, in the Database module, the Locate time is logarithmic in the total number of keys, and, because the Locate requests are for user addresses, the relevant set of keys is that of all user addresses, which number $U \times N$.

The average Locate time is significantly more sensitive to the number of provider addresses per node, P: in our measurements, the variation versus P was linear; about 150 μsec for every unit increase in P. There are two aspects of our design that significantly contribute to this value. First, the list of provider addresses, which is returned by the *Database* module to the *Server Protocol* module, must be pruned of potential duplicates. Second, a self-contained copy of the resulting list must be created. This involves memory allocations, which are time-consuming. Although 150 μsec per provider address is relatively long, we expect that in practice the returned list of provider addresses will rarely have more than three elements.

5. Conclusion

The MPTN Address Mapper Architecture and an implementation thereof were presented. It was shown that the architecture satisfies the requirements of address mapping in heterogeneous networking environments, in that it supports a universal address format, a dynamic address database, speedy handling of requests, group-address, and wildcard operations, as well as scalable implementations. The implementation of the Address Mapper was shown to be highly structured and to reuse a large amount of code between the Server and Client Libraries. The performance of this implementation was also presented.

Acknowledgments

It is a pleasure to acknowledge the fruitful cooperation with many colleagues during the design of the MPTN architecture, including Diane Pozefsky, Roger Turner, Sandy Hobgood and Doug Dykeman, to name only a few. We are also indebted to our colleagues Pitro Zafiropulo for implementing the formatting package, and Liba Svobodova for providing valuable comments on the initial draft of this paper.

References

[1] J. Auerbach. TACT: A protocol conversion toolkit. *IEEE Journal on Selected Areas in Communication*, 8(1):143–159, January 1990.

[2] Ernst W. Biersack. Annotated bibliography on network interconnection. *IEEE Journal on Selected Areas in Communication*, 8(1):22–40, January 1990.

[3] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(3):23–29, June 1989.

[4] D. Comer. *Internetworking with TCP/IP. Principles, Protocols and Architecture.* Prentice-Hall, New Jersey, first edition, 1989.

[5] W. Doeringer, D. Dykeman, G. Bollella, E. Chen, D. Chung, A. Edwards, J. Mathew, D. Pozefsky, Sarkar S., R. Turner, K. Tracey, D. Poirier, J. Fetvedt, and S. Hobgood. Multiprotocol transport networking: A software declaration of application independence. *IBM Systems Journal*, 34(3):???, August 1995.

[6] W. Doeringer, D. Dykeman, M. Kaiserswerth, B. Meister, H. Rudin, and R. Williamson. A survey of light-weight transport protocols for high-speed networks. *IEEE Transactions on Communications*, 38(11):2025–2039, November 1990.

[7] W. Doeringer, G. Karjoth, and M. Nassehi. Routing on longest-matching prefixes. *Submitted to IEEE Transactions on Networking*, 1995.

[8] IBM. *NetBIOS Application Development Guide*. Document S68X-2270.

[9] M. Lottor. Domain administrators operations guide. RFC 1033, NIC, 1997.

[10] P. Mockapetris. Domain names - concepts and facilities. RFC 1035, NIC, 1987.

[11] L. Morales and P. Hasse. IP to X.121 address mapping for DDN. RFC 1236, NIC, 1991.

[12] D.M. Ogle, K.M. Tracey, R.A. Floyd, and G. Bollella. Dynamic protocol selection for socket applications. *IEEE Network Magazine*, 7(3):48–57, May 1993.

[13] Radia Perlman. *Interconnections: Bridges and Routers*. Addison-Wesley, Reading, Massachusetts, first edition, 1992.

[14] R. Watson and S. Mamrak. Gaining efficiency in transport services by appropriate design and implementation choices. *ACM Transactions on Computer Systems*, 5(2):97–102, May 1987.

[15] C. Weider and J Reynolds. Executive introduction to directory services using the X.500 protocol. RFC 1308, NIC, 1992.

[16] X/Open. *Multiprotocol Transport Networking (XMPTN) Access Node*, 1995. X/Open Specification P408, ISBN 1-85912-040-7.

[17] X/Open. *Multiprotocol Transport Networking (XMPTN) Address Mapper*, 1995. X/Open Specification P407, ISBN 1-85912-039-3.

[18] X/Open. *Multiprotocol Transport Networking (XMPTN) Formats*, 1995. X/Open Specification P448, ISBN 1-85912-043-1.

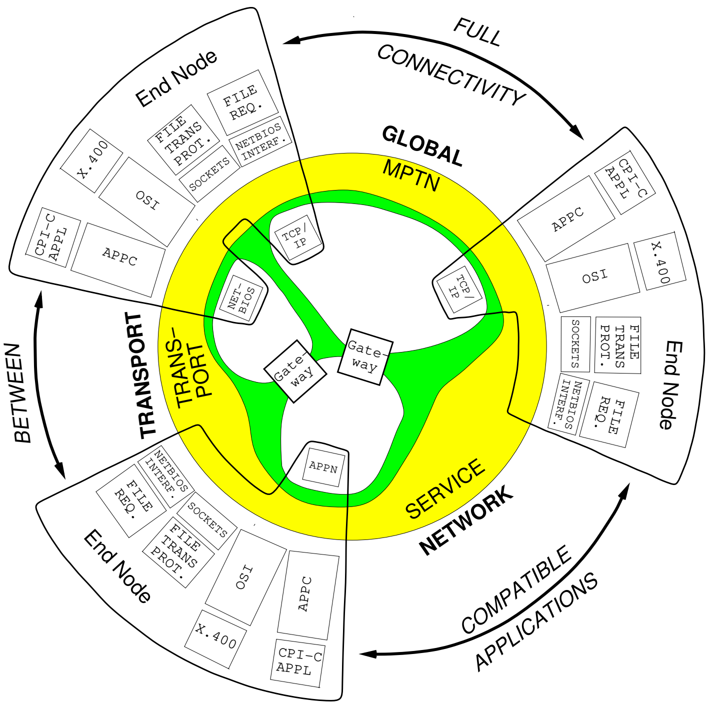


Figure 1: MPTN provides global connectivity.

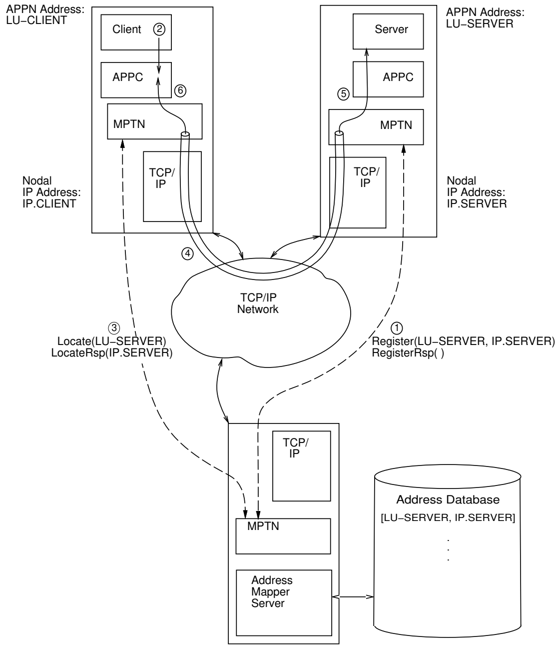


Figure 2: MPTN Address Mapping Principles.

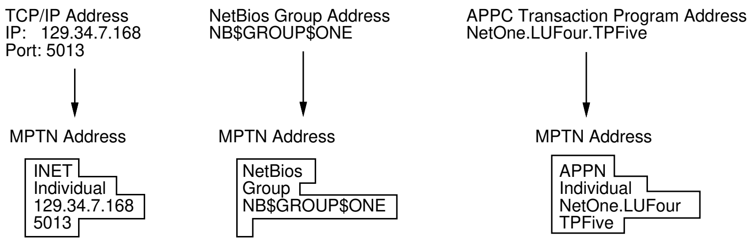


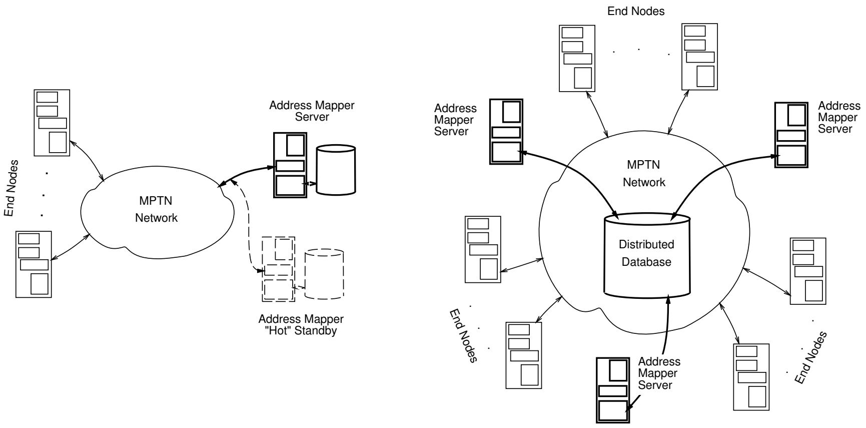
Figure 3: MPTN Address Format.

Query Request	
Field name	*Bytes*
Header Prefix	*4*
command code	1
processing specification	1
command length	2
Transaction Identifier	*2 . . . 9*
Return code	*3*
return code value	1
modifier	2
MPTN node address of requestor	*5 . . . 258*
UserAddress to be resolved	*5 . . . 512*

Addresses are in MPTN format.

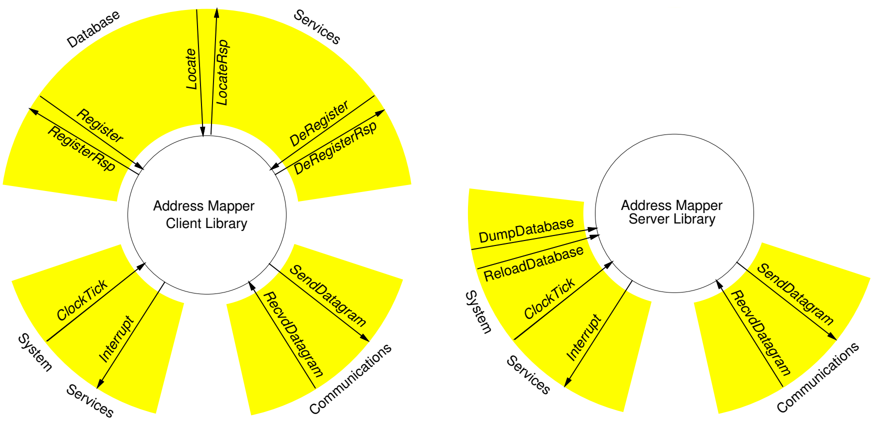
Query Response	
Field name	*Bytes*
Header Prefix	*4*
command code	1
processing specification	1
command length	2
Transaction Identifier	*2 . . . 9*
Return code	*3*
return code value	1
modifier	2
MPTN node address of sending server	*5 . . . 258*
Original UserAddress	*5 . . . 512*
ProviderAddress to reach UserAddress	*5 . . . 512*
. . .	. . .
ProviderAddress to reach UserAddress	*5 . . . 512*

Figure 4: Line-flow formats of Locate Request and Response.



(a) Medium-sized MPTN Network. (b) Large MPTN Network.

Figure 5: Address Mapper Deployment Examples.



(a) *Client Library.* (b) *Server Library.*

Figure 6: MPTN Programming Interfaces.

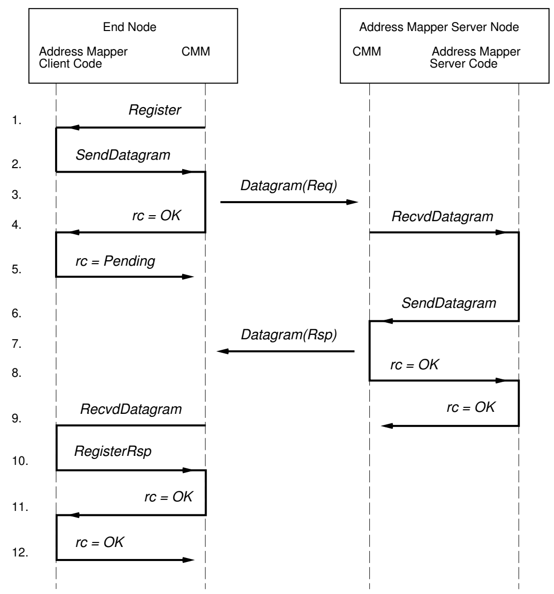


Figure 7: MPTN Execution Flow Example.

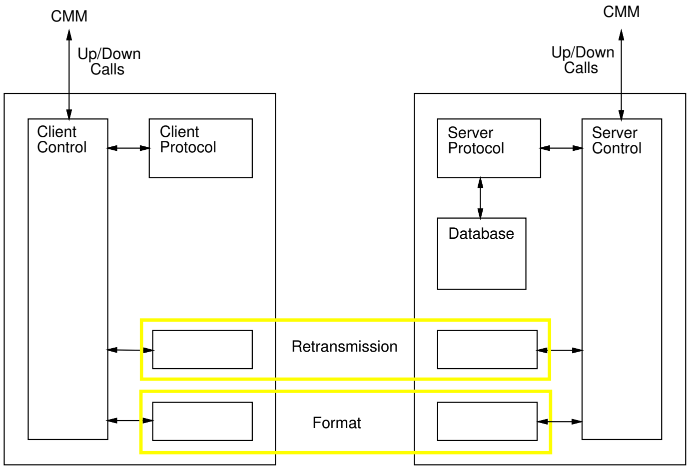


Figure 8: MPTN Module Diagram.