SICS Technical Report
T92:06

ISRN : SICS-T--92/06-SE
ISSN : 1100-3154

On Porting VMTP to the x-kernel

by

Kent Lundin and Per Gunningberg

Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, SWEDEN

On porting VMTP to the x-kernel

Kent Lundin and Per Gunningberg

Swedish Institute of Computer Science

Box 1263

S-164 28 Kista

Stockholm, SWEDEN

June 1992

SICS Technical Report T92:06

Abstract

This report describes our port of Stanfords transaction protocol VMTP from the BSD 4.3 UNIX kernel implementation to the x-kernel operating system from University of Arizona. The BSD UNIX kernel dependent functions in the VMTP implementation are identified. It is described how we have replaced these functions with corresponding parts from the x-kernel. In particular we describe how BSD UNIX "sockets" are replaced with x-kernel parts and how the BSD UNIX buffer system, "mbufs", is moved into user space. The ported version is the so called minimal implementation of VMTP. It only runs in the simulator of the x-kernel.

1. Introduction.

In the Swedish MultiG program there is a need for a transport service for multimedia applications and high speed networks. Stanford´s transport protocol VMTP belongs to a new generation of protocols for high speed networks. We wanted to evaluate how well VMTP can meet the requirements of multimedia applications and how efficiently it can be implemented. There exists a few publically available implementations of VMTP[4].

Many of the MultiG multimedia applications will run under the Mach operating system. Therefore it was desirably to evaluate a version of VMTP which runs under Mach. Unfortunately, there existed no such run-able version. Hence, we decided to port an existing version to Mach. The most straightforward approach is to port the UNIX BSD kernel version to the UNIX BSD kernel in Mach. This approach has also been attempted at the Mach development group but it was never completed. We decided to port the BSD UNIX implementation of VMTP to the x-kernel for three reasons. First, we considered the BSD environment too slow for our anticipated applications. Second, the Mach development group has decided to use the x-kernel from University of Arizona as their network protocol environment instead of the BSD UNIX[10]. Third and last, we believe that the x-kernel is more suitable as a testrig for experiments on efficient implementation of protocols than BSD UNIX[5].

Our porting strategy was to use as much of the BSD UNIX implementation as possible and to successively replace nonportable parts with corresponding x-kernel parts. The BSD UNIX code is written in the programming language C and x-kernel supports C programs so the main task was to find a mapping from UNIX specific kernel primitives (like mbufs, sockets etc.) to x-kernel primitives. We will in this paper present our stepwise strategy for doing this. More details about the port can be found in [6].

A subset of VMTP, the so called "Minimal implementation" has been ported. It consists of the major parts of the VMTP state machine, the request/response parts. Still to be ported are the timer management, multicasting facility, message forwarding and the security parts. The UNIX socket interface code for VMTP has been replaced with the corresponding x-kernel interface. The UNIX buffering system, mbufs, has been emulated in the x-kernel. This subset has been tested in the x-kernel simulator (under SunOS). Since this implementation uses UDP instead of IP as the underlying communication service it can not yet run outside the simulator. Other parts that need to be ported or added are: an interface to IP instead of UDP, a timer management interface to x-kernel, and to replace mbufs with x-kernel buffers.

The next section gives an brief overview of VMTP and the x-kernel. The existing implementation of VMTP in UNIX BSD is described as well as the main differences between the UNIX and the x-kernel protocol environments. The following section describes our porting strategy. It goes into details on how we ported sockets and mbufs. Section 5 discusses directions for future research and lists parts that still have to be ported.

2. VMTP and its UNIX implementation

VMTP, Versatile Message Transaction Protocol, was originally developed for Stanford's V-System[2]. It is a transport protocol, specifically designed to support the transaction model of communication, as exemplified by the remote procedure call (RPC) paradigm. The full function of VMTP include support for security, real-time, asynchronous message exchanges, streaming, multicast and message forwarding.

VMTP has some attractive functionalities which could be used for high performance networks and multimedia applications[4]: It uses rate control for flow control instead of the common window mechanism, which causes a burst behavior. Voice and video applications are sensitive to the delay variations in packet transfer time caused by these bursts. In rate control, an artificial delay is inserted between packets instead of sending them "back-to-back". The length of this delay is adjusted according to network and receiver capacity parameters. Another attractive functionality for multimedia is VMTP's multicasting facility. A third attractive function of VMTP is the possibility to select error recovery strategy depending on user application needs. For example a voice transmission can accept sporadic bit errors but few or none retransmissions due to the extra delay they cause.

In overview, VMTP provides a transport service service via message transactions. A transaction consists of a request message sent by a client, to a server (or group of servers) followed by a response message from the server (could also be several for a group or zero for datagram service) to the client. The response message will acknowledge the request message. A response message could be positively acknowledged if specified. Timers are used to detect missing responses and inactive partners.

Figure 1 describes the major parts and the information flow of the BSD 4.3 UNIX VMTP protocol implementation entity. It has two sides, an outbound side for messages to be sent on the network (left hand side in the figure) and an inbound side. The entity could operate either as a client or a server. Requests to send messages arrive from the user or are generated by the entity itself. At the outbound side, a user application program calls the protocol via the UNIX socket system calls, augmented with parameters for VMTP specifics. VMTP state machine interfaces the socket system via the VMTP usrreq module. Depending on whether it is a request or a response message, alternative paths are chosen in the entity. Also, if special services are

required such as security, corresponding modules will be called. Eventually the message will be fully assembled and forwarded either to the IP protocol or, if the communicating entity is on the same machine, to the input queue of the inbound side as illustrated in the figure.

At the inbound side there are corresponding modules for client and server sides. If an incoming message demands an explicit acknowledgement the outbound entity is called. Otherwise, the message will be written to the socket interface and a user program wakeup will be generated.

For the port to the x-kernel environment the socket interfaces both for inbound as well as for the outbound direction have been be changed. Accordingly, the interfaces to lower protocols have been changed to corresponding interfaces in the x-kernel.

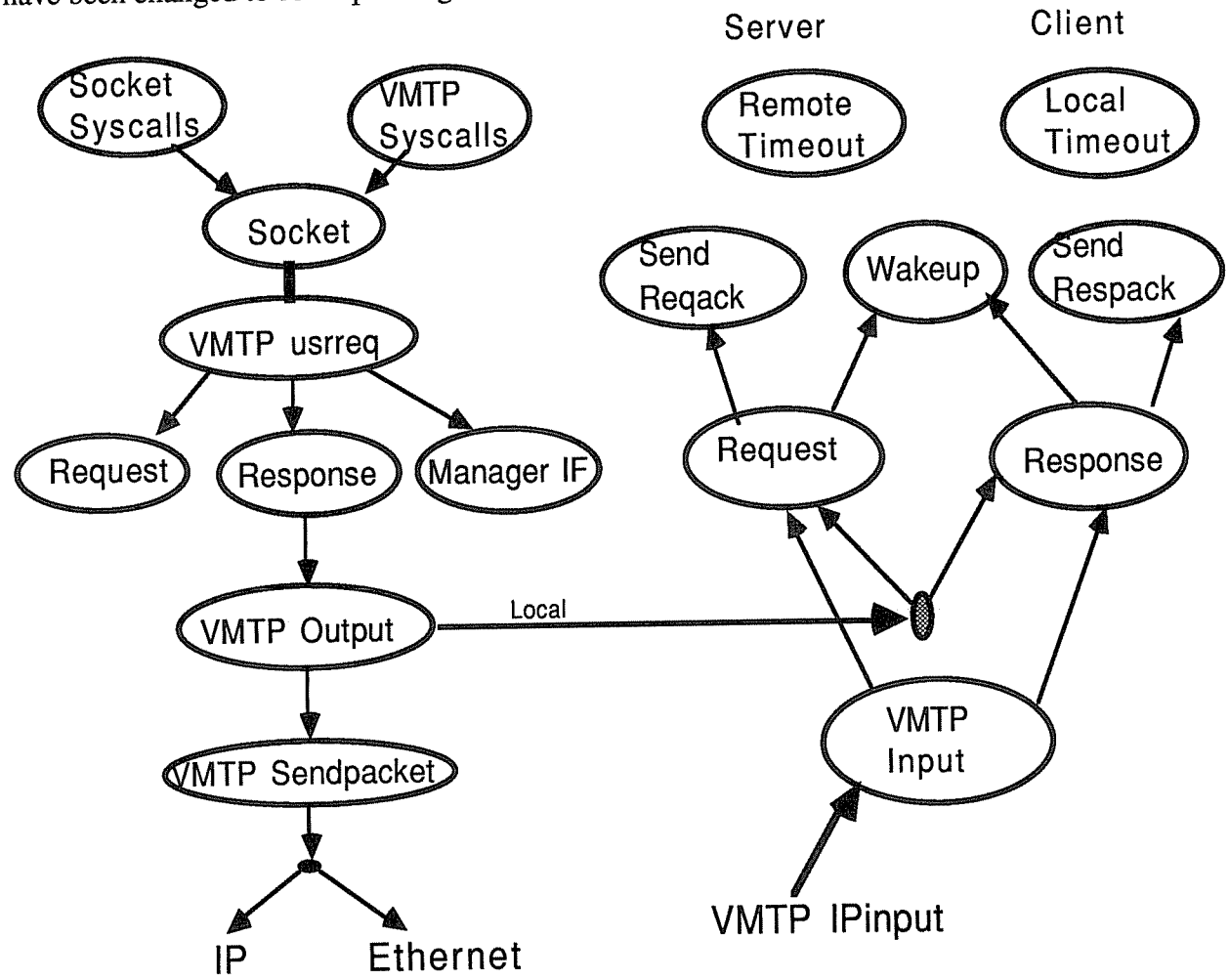

Figure 1. VMTP Flow Graph

3. The x-kernel

The x-kernel is an operating system kernel. It is developed specifically as an architecture for constructing and composing network protocols[13]. It runs on Sun3 and Sun SPARC workstations among others. It is configurable, supports multiple address spaces and has light-weight processes for fast context switches. There is also a x-kernel simulator for the development and testing of protocols. The simulator runs as a user program in UNIX and hence all protocol development can be done with available UNIX user programming tools. After a protocol has been developed in the simulator, it is compiled and linked with the x-kernel and other protocols. The combination and the relationship between protocols is defined at the

time the kernel is configured with a dependency graph. In other words the x-kernel can be configured with only the protocols needed by a particular application.

The x-kernel has three attractive features for protocol design and implementation. First, it defines a set of abstractions for encapsulating protocols. With these, uniform protocol interfaces can be designed. These abstractions make it straightforward to combine protocols from different stacks. Second, the x-kernel provides efficient interaction between abstractions. Or more concrete, an x-kernel layers is light-weight, ie.. it costs only one procedure call to pass a message from a high-level protocol to a low-layer protocol and vice verse. Third, the x-kernel provides efficient implementation of the operating system routines that are frequently called by protocol implementations. For example, the x-kernel has routines for buffer management and event handling. Performance measurements on protocols implemented in the x-kernel show that their performance are competitive with other more ad hoc implementations in for example UNIX and Sprite[8].

X-kernel structures
The object-oriented infrastructure for implementing and composing protocols is based on three abstract communication objects, **protocols**, **sessions** and **messages**. Each protocol is encapsulated into a single **protocol** object and a collection of **session** objects. A **session** is an instance of a protocol created at runtime and corresponds to a connection[9]. It interprets **messages** and maintains state information associated with a particular connection. Hence a session has similarities with a UNIX socket. The **protocol** objects demultiplex messages received from the network to one of their **sessions** with the **demux** operation. A **protocol** object corresponds to the UNIX triple *<domain, type, protocol>*.

The x-kernel associates a process with each message rather than with a protocol or a layer. This process "shepherds" messages through a series of protocols to/from the kernel boundary and the device driver according to message connection identifiers, protocol type and associated state.

4. UNIX dependent parts of VMTP

The BSD UNIX version of VMTP has several UNIX dependent parts which have been ported. They include UNIX mbufs, the socket interface, the IP interface and system calls, like Wakeup (call to the user process). The major design and implementation effort consisted of replacing these parts with corresponding x-kernel constructs. We will in this section describe how the VMTP socket interface and mbufs are ported.

We used x-kernel's simulated environment when we ported the VMTP protocol. The simulator provides a UDP datagram service for communication outside the simulator, i.e. with other x-kernel simulators. Protocols in the simulator can only talk to their peers in other simulators since a non standard addressing scheme is used. The UDP interface was used instead of the normal IP interface of VMTP. VMTP's IP packets are thus encapsulated within UDP datagrams.

The protocol interfaces
We implemented in x-kernel an interface function which handles the interfaces between the VMTP state machine and the higher level and the lower level protocols. The VMTP functions which were called upon via socket system calls are now called via this function. The VMTP socket interface has been replaced with an x-kernel interface. The BSD UNIX create, open and close socket calls has been replaced with **sessions** create, open and close. Furthermore the function controls the "pushes" and "pops" of messages through **sessions** and demultiplex messages with calls to **demux** objects. With a **push** operation a message is sent down to a lower layer protocol and with a **pop** operation the message is sent to the protocol above. The **demux** operation takes a message as an argument and passes the message to one of its

sessions. Since a user in x-kernel is masqueraded as a protocol, it is sufficient with a simple **push** operation to pass a message from the user to VMTP.

UNIX BSD 4.3 mbufs
UNIX BSD 4.3 *mbufs* data structures are created to hold a whole or a part of a message. At configuration time kernel address space is reserved for these mbufs, ie. memory buffers. In the kernel there are routines which manage these buffers. An mbufs is 128 bytes long with a 112 bytes data space. Larger data spaces (up to 1 Kbyte) may be associated with a mbuf by referencing a page (also called a cluster, 1024 bytes data area). An mbufs structure has a length field which indicates the number of bytes of valid data at an offset. The offset and the length fields allow mbufs routines to trim data efficiently at the start or end of an mbufs. Multiple mbufs can be linked together to hold an arbitrary number of data items. The linkage is accomplished with the m_next field. There is also a m_act field which can be used to link objects constructed from chains of mbufs into lists of objects. The m_type field is used for two purposes, to maintain information about storage usage and to distinguish some optional components when mbufs are linked at socket data queues.

BSD UNIX has two free lists, one for mbufs structures and one for clusters. If more mbufs or clusters are needed, more space is dynamically created and linked into the free list.

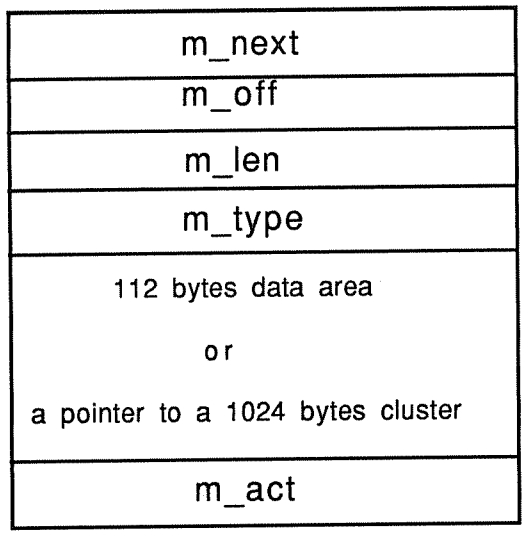

Figure 2. The mbuf structure

mbuf port to x-kernel
Our porting strategy was to rewrite the mbufs and their routines into corresponding parts in the x-kernel in several steps. Our first step was to move the whole VMTP UNIX kernel implementation into user space in order to find all kernel routine dependencies. The second step was to emulate the mbufs and the mbufs routines and macros in this user space. The third was to move this user space implementation of VMTP to the x-kernel environment and to get it to run with mbufs. The fourth and last step is to replace the mbufs and their routines with the x-kernel corresponding structures and routines. So far, we have accomplished steps one to three.

One of the problems in step two was how to allocate user address space memory so the UNIX routines dtom and cltom (data-to-memory, cluster-to-memory) could operate on mbufs and clusters as in kernel space. These functions require that mbufs are allocated at an even 128 bytes address and that cluster are allocated at an even 1024 bytes address. The argument to dtom and cltom functions is an address into the mbufs data area or the cluster area and they return pointers to the mbufs headers. This is straightforward when the mbufs are aligned on

128 or 1024 byte boundaries. The solution to this problem was to write two separate mbufs allocation routines, one for mbufs and one for cluster. The current allocation routine for mbufs allocates an memory area, sets the free-list pointer at an even 128 bytes address and organizes the rest of the allocated space into a linked list of mbufs. The allocation routine for cluster is almost identical except that the free-list pointer points to an even 1024 bytes address. We keep track of these areas with two global pointers (mbuf_alloc and cluster_alloc).

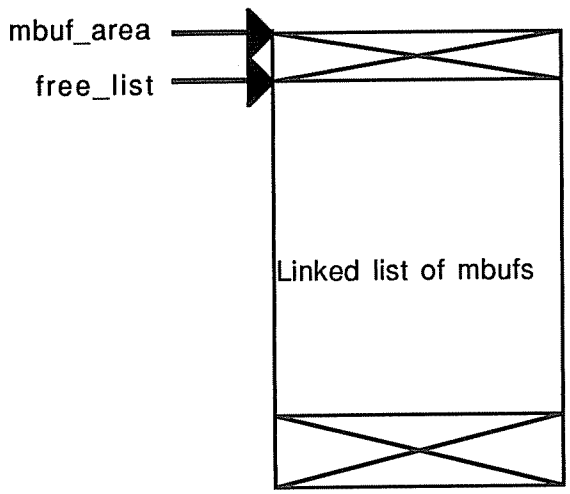

Figure 3. Allocated space for mbufs

5. Directions for future work

mbufs
Step four, i.e. to replace mbufs with x-kernel message structures still needs to be done. Some of the mbufs are used for dynamic storage of connection states and local VMTP states. There is no need to use mbufs for these states, they were used because of their dynamic conveniences. Therefore other structures should be allocated for these states. The VMTP structures that are stored in mbufs are vmtppool, vmtplpool and vmtpesr.

UDP socket to IP session
Our current implementation of VMTP in the x-kernel has a socket interface to BSD UNIX´s UDP. It has to be replaced with an x-kernel IP session interface. The data written to the socket is now a character string and it has to be changed into an x-kernel message. The IP protocol in x-kernel also needs to be modified since VMTP uses a version which supports multicasting.

Message Transactions
To be able to use all the functions of VMTP (for instance multicasting), it is necessary to implement VMTP's IP in the x-kernel. It might also be necessary to make some changes in the alternative sending routines of VMTP when data is read from continues/noncontinues memory (see vmtp_output.c).

Timer handling
The VMTP timer handling routines have to be replaced with the x-kernel timer handling routines. In current version we do not use the UNIX kernel timer routines.

6. References

[1] D. R. Cheriton. VMTP: *A Transport Protocol for the Next Generation of Communication Systems*. SIGCOMM '86, ACM. Aug. 1986.

[2] D. R. Cheriton. VMTP: *Request-Response and Multicast Interprocess Communication in the V Kernel*. Computer Science Dept, Stanford

[3] D. R. Cheriton. VMTP: Versatile Message Transaction Protocol *Protocol specificaton* Internet Request for comments RFC-1074, SRI Network Information Centre, Feb. 1988.

[4] D. R. Cheriton, Carey L. Williams. *VMTP as the Transport Layer for High-Performance Distributed Systems*. IEEE Communication Magazine. June 1989.

[5] B. Pehrson, P. Gunningberg, S. Pink. *MultiG - A Research Program on Distributed Multimedia Applications and Gigabit Networks*. IEEE Network, January 1992

[6] L Elmstedt, K Lundin: *Implementing the VMTP protocol under x-Kernel*. A thesis work. Dec. 1990.

[7] N. C. Hutchinson, L. Peterson. *Design of the x-kernel*. In Proceedings of the 1988 Symposium on Communication Architecure and Protocols, Aug 1988.

[8] N.C. Hutchinson, L. Peterson, M. Abbot, S. O'Malley. *RPC in the x-Kernel: Evaluating New Design Techniques*. Proc. 12th ACM Symp. Operating System Principles, Dec 1989

[9] N. C. Hutchinson, L. Peterson. *The x-kernel: An Architecture for Implementing Network Protocols*.

[10] D. Julin, Personal Communication, December 1990.

[11] Erik Nordmark, D. R. Cheriton. *Experiences from VMTP: How to achieve low response time*. March 1989.

[12] Erik Nordmark. *VMTP Implementation Notes*. Stanford University, June 1987.

[13] L. Peterson, N. C. Hutchinson, S. O'Malley, H. Rao. *The x-kernel: A Platform for Accessing Internet Resources*. IEEE Communication Magazine. May 1990.

[14] L. Peterson, N. C. Hutchinson, S. O'Malley. *x-Kernel Programmer's Manual(Version 3.0) TR 89-29*.