View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Washington University St. Louis: Open Scholarship

Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCS-89-13

1989-05-01

Axon: Network Virtual Storage Design

James P.G. Sterbenz and Gurudatta M. Parulkar

This paper describes the design of network virtual storage (NVS) in the Axon host communications architecture for distributed applications. The Axon project is investigating an integrated design of host architecture, operating systems, and communications protocols to allow applications to utilize the high bandwidth provided by the next generation of communications networks. NVS extends segmented paged virtual storage management and address translation mechanisms to include segments located across an internetwork. This provides the ability to efficiently use the shared memory paradigm in non-local environments, as well as the support for a very high speed end-to-end data path between demanding applications... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation

Sterbenz, James P.G. and Parulkar, Gurudatta M., "Axon: Network Virtual Storage Design" Report Number: WUCS-89-13 (1989). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/726

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/726

Axon: Network Virtual Storage Design

James P.G. Sterbenz and Gurudatta M. Parulkar

Complete Abstract:

This paper describes the design of network virtual storage (NVS) in the Axon host communications architecture for distributed applications. The Axon project is investigating an integrated design of host architecture, operating systems, and communications protocols to allow applications to utilize the high bandwidth provided by the next generation of communications networks. NVS extends segmented paged virtual storage management and address translation mechanisms to include segments located across an internetwork. This provides the ability to efficiently use the shared memory paradigm in non-local environments, as well as the support for a very high speed end-to-end data path between demanding applications such as scientific visualization and imaging.

AXON: NETWORK VIRTUAL STORAGE DESIGN

James P. G. Sterbenz
Gurudatta M. Parulkar

WUCS-89-13

May 1989

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

ABSTRACT

This paper describes the design of *network virtual storage* (NVS) in the Axon host communications architecture for distributed applications. The Axon project is investigating an integrated design of host architecture, operating systems, and communications protocols to allow applications to utilize the high bandwidth provided by the next generation of communications networks. NVS extends segmented paged virtual storage management and address translation mechanisms to include segments located across an internetwork. This provides the ability to efficiently use the shared memory paradigm in non-local environments, as well as the support for a very high speed end-to-end data path between demanding applications such as scientific visualization and imaging.

Revised October 24, 1989 to exclude introductory Axon material now in [StPa89a] and to incorporate minor technical updates. Minor revision April 3, 1990. Abridged versions of this paper appear in Computer Communication Review [StPa90a] and the 10th ICDCS Proceedings [StPa90c].

James Sterbenz is on leave of absence from IBM Corporation at Washington University in St. Louis.

AXON: NETWORK VIRTUAL STORAGE DESIGN

James P. G. Sterbenz
jps@wucs1.wustl.edu
+1 314 726 4203

Gurudatta M. Parulkar
guru@flora.wustl.edu
+1 314 889 4621

1. Introduction

It is being recognised that high performance interprocess communication (IPC) across a network can benefit from the ability to use the shared memory paradigm. We have proposed a new communication architecture for distributed applications called **Axon** [StPa89a, StPa90b] whose principal motivation is to provide high performance IPC in the future generation of internetwork, which we refer to as the *very high speed internetwork* (VHSI) [Pa90]. The significant features of Axon are: [1] an integrated design of host and network interface architecture, operating systems, and communication protocols; [2] a network virtual storage facility which includes support for virtual shared memory across networks; [3] a high performance, lightweight object transport facility which can be used by both message passing and shared memory mechanisms [StPa89b]; [4] a pipelined network interface which can provide a high bandwidth low latency path directly between the VHSI and host memory [St90].

This paper describes the Axon *network virtual storage* (NVS) design, and is organised as follows: Section 2 provides a brief overview of the Axon host communications architecture for distributed systems. Section 3 introduces virtual storage concepts and implementation. Section 4 describes the NVS design, by extending the mechanisms for a segmented paged virtual store to include segments located across an internetwork. Section 5 describes virtual storage management policies, including alternatives and tradeoffs involved in NVS. Section 6 describes other relevant work, and Section 7 is the conclusion. Appendix A contains the specification of the NVS address translation data structures.

2. Overview of the Axon Architecture

This section provides an introduction to the Axon architecture. First, IPC primitives are discussed within the framework of the VHSI environment. Then, a brief description is presented for significant Axon architectural components: system level IPC support, the transport protocol (ALTP-OT), host and network interface architecture, and the communications processor (CMP).

2.1. IPC in the Axon architecture

IPC is typically supported as either a shared variable or message passing paradigm. Shared variable IPC is characterised by the use of *read/write* (r, w) primitives to shared data structures. Message

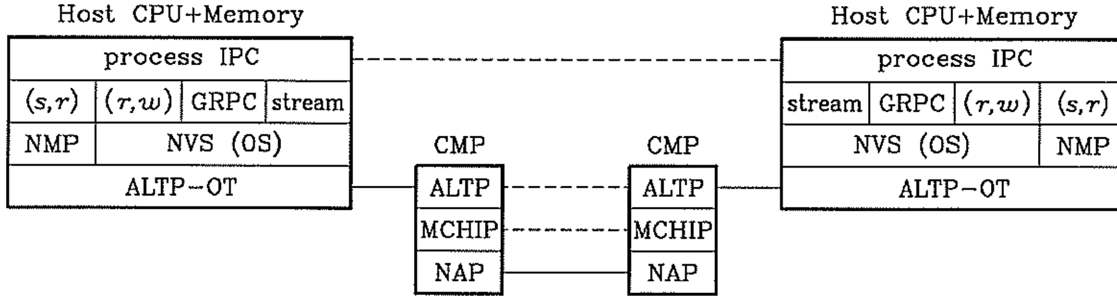


Figure 1: Logical Axon Protocol Hierarchy

passing IPC is characterised by the use of *send/receive* (s, r) primitives, and requires explicit synchronisation by the processes communicating. A generalised form of remote procedure call (GRPC) is supported, in which the location of the procedure, data, and execution are all arbitrary. Additionally, *segment streaming* is supported, which transports multiple segments at high bandwidth with no explicit *per* segment request.

A *logical* view of the Axon protocol hierarchy is presented in Figure 1. It is important to note that this is a logical view of function only, and does not imply that strict layering (in the ISO-OSI sense) is adhered to.

At the system level, the shared memory mechanism for IPC across the VHSI is implemented by NVS (network virtual storage). This can be utilised by an application which references segments that are non-local, through the facilities provided by GRPC, or by the use of segment streaming. Support for message passing IPC is provided by a network message passing interface (NMP), which provides the support for invocation of the appropriate message-based transport protocol operations. The transport mechanism is supplied by an *application-oriented lightweight transport protocol* (ALTP) tailored for the class of applications using IPC *object transfer*; this type is ALTP-OT. ALTP-OT resides as a set of software modules in the host system, and as hardware in the communications processor (CMP). The underlying internet/network layers of function is provided by a *multipoint congram-oriented high-speed internet protocol*† (MCHIP) [MaPa89, Pa90], and *network access protocols* (NAP). The GRPC and segment streaming IPC paradigms will now be described.

Generalised remote procedure call. A typical (local) procedure call is of the form:

call ⟨*procedure-name*⟩ ⟨*parameters*⟩

which indicates that control will be transferred to the procedure named *procedure-name*, with input parameters passed on invocation and output parameters returned on termination. Axon supports a more general form of procedure call, in which the code and data segments can be located on arbitrary and independent hosts, with execution specified for an arbitrary host, referred to as *generalised remote procedure call* (GRPC). Conventional RPC [BiNe84] is a restricted form of GRPC.

Using GRPC, a procedure is invoked which resides on either the local or remote host, is executed on any host, with data parameters from any host:

call ⟨*procedure-name*⟩ {local | remote | ⟨*host-name*⟩} ⟨*parameters*⟩

†A congram combines the desirable features of a datagram with those of a (soft) connection. For the purposes of this paper, it can be thought of a connection with the added attributes of rapid setup and survivability in the presence of network failures.

The symbolic specification of *procedure-name* is bound by the local system to a virtual address, which is either on the local system, or on a remote host. The mechanism for accomplishing this is provided by NVS, and will be described later in this paper. The second parameter specifies whether the execution of the procedure will take place on the local, remote (location of *procedure-name*), or other specified *host-name*. The parameters are also bound to virtual addresses which may be local or remote.

A number of useful call semantics arise from this model, some of which are not incorporated into conventional RPC. Some of these are:

type	program	data	execution
program server	remote	local	local
conventional RPC	remote	local	remote
database access	local	remote	local

Two significant cases arise out of the possible GRPC call types, depending on whether the procedure is to execute on the local host or on a remote host. This determines which ALTP-OT call is used to perform the transport.

If the procedure is to execute locally, it does so under the context of the calling process. Any segments (code and data) needed for the procedure invocation which are not locally resident are fetched across the network with the get-segment ALTP-OT operation. This is done automatically by the NVS implementation, based on the resolution of *procedure-name* and *parameters*.

If the procedure is to be executed remotely, the code segment (along with any data segments part of the same segment access group) will be sent to the remote host for execution with the remote-execute ALTP-OT operation. The corresponding ALTP-OT instantiation at the remote end is then responsible for issuing the required system calls to create a process context for the procedure execution. Any other required data segments from other hosts will be gotten at the remote host using the usual NVS mechanisms and get-segment operations.

Segment streaming. The special demands of high performance visualisation and imaging applications motivate an additional IPC paradigm. Two mechanisms are provided to transfer *segment streams* at high bandwidth with low setup overhead.

The first provides for the *repetitive* transmission of segments, sending them across an open connection at time intervals specified by the program (*interval synchronised*), or whenever an appropriate program call is executed (*program synchronised*). Examples of repeated transfer include the transmission of successive image frames for motion video applications, and simulation data that is periodically updated to be sent to a graphics workstation for local visualisation.

The second type of segment streaming allows for the *sequential* (interval or program synchronised) transfer of segments in a segment group. For example, a simulation that produces groups of data points can transmit automatically when each additional group is complete, by placing the appropriate program calls after data group generation.

The performance advantage of segment streaming is that a single ALTP-OT call performs the request for all of the segments, and each segment can be transmitted when ready across a VHSI connection without the latency of request or setup.

2.2. System level IPC support

The system level support for the various application level IPC paradigms is provided by two components: NVS and NMP. NVS is the system shared memory interface for shared variables, GRPC, and segment streaming IPC. NVS is described in detail in Sections 4 and 5. NMP is the system level message passing interface. It performs relatively straight-forward transformations from application level primitives (*e.g. send* and *receive*) to ALTP-OT message passing calls (*e.g.* get-message and send-message).

2.3. Transport protocol (ALTP-OT)

At the transport level, the VHSI model is best supported by a set of simple *application-oriented lightweight transport protocols* (ALTP) for various classes of applications [BhSt88, PaTu90, StPa89b]. These transport protocols can have their *critical path* function implemented in VLSI hardware. The critical path consists of the data path and routine control functions allowing data to flow once a transport operation has begun. By optimising the critical path functions and processing multiple packets in a single transport level operation, the *per* packet processing can be performed in real time at the full VHSI data rate. For the protocol to be efficiently implemented in hardware, the protocol, hardware design, and host operating system should be well integrated.

The ALTP type that is used by NVS is designed to support IPC by the transfer of objects (with primary consideration in supporting NVS segments), referred to as ALTP-OT. ALTP-OT uses rate based flow control, where the rate specification consists only of parameters important to IPC, and efficient error control streamlined to include only what is necessary for object transfer. The various error conditions are handled by ALTP-OT as follows: duplicate packets are discarded; corrupted packets are discarded with retransmission requested; missing packets are detected by the expiration of a timer with retransmission requested; packet sequence is ignored since the packets are placed directly in the proper location of the target store (*sequence by placement*).

Information is transferred throughout the VHSI in packets. A group of packets corresponding to a single ALTP-OT semantic action is a *super-packet*, consisting of an initial control packet (which may also contain data), and optionally followed by data packets. In the case of a segment transfer (*e.g.* to satisfy a remote segment fault), a super-packet consists of the entire segment. The correspondence between host and network objects, and the resulting correspondence of control (*e.g.* segment fault resolution and super-packet processing) provides substantial performance benefits, in terms of reduced overhead of data buffering/reformatting and control synchronisation.

The ALTP-OT requests include connection establishment/termination, segment/page/message transfer, and packet retransmission. ALTP-OT is described in detail in [StPa89b].

2.4. Host and network interface architecture

High performance computer systems typically consist of one or more central processors (CPU), which communicate with memory banks and I/O processors through an interconnection network. Communication is typically handled by front-end communications processors or network interfaces, which use the I/O interface to the host system. In the VHSI environment, it is necessary to provide high bandwidth low latency data paths directly to memory, motiving new host architectures. Two host architecture configurations are defined for the Axon architecture:

Interconnect interface architecture (IIA). The first host configuration gives the CMP (communications processor) a relationship to the system similar to that of I/O processors, thus interfacing the CMP directly into the processor–memory interconnection network. This is referred to as *interconnect interface architecture* (IIA). In addition, an interconnection between CMPs and I/O processors should be provided to allow direct, high-speed transfers between the network and I/O controllers or devices (which provides the path to auxiliary or secondary storage). Axon only imposes the requirement that the interconnection be rich enough to allow the added CMP connections, and high enough performance to sustain the additional VHSI communication traffic.

Memory interface architecture (MIA). The second host configuration interfaces the CMP to a special multi-ported *communications memory module* (CMM), which is referred to as *memory interface architecture* (MIA). In this case, the CMM has a conventional random access port which appears like any other memory bank to the processor–memory interconnect. The other ports are high speed serial access interfaces to the CMP. The design of the CMM is similar in concept to VRAM (video-RAM), and requires that the physical address space of the system be partitioned between conventional and communications memory. More information on Axon host architecture and the network interface is presented in [St90].

2.5. Communications processor (CMP)

The Axon architecture interfaces the CMP directly to the processor or memory, specifically as a host–network interface processor. On the network interface side, the CMP must be capable of receiving and transmitting packets at the full network data rate. On the host side, the CMP must either interface to the processor–memory interconnect or the CMM, depending on the host architecture (IIA or MIA respectively).

The primary design consideration of the CMP is to serve as the network interface to the VHSI, as part of an end-to-end pipelined data path between application memory spaces. Thus the CMP must have the ability to perform critical path functions in real time with no packet buffering, and incorporate the necessary function in VLSI. This may be realised by organising the CMP as a pipeline, dynamically reconfigurable based on the ALTP type and options for a particular connection. The pipeline organisation allows packets to be processed while moving at the VHSI interface data rates. Greater detail on the CMP design is presented in [St90].

3. Virtual Storage Models

This section introduces virtual storage techniques in the context of extending virtual addressing to the VHSI environment. The purpose is to provide background information, as well as to introduce notation and ideas that will be built upon in the following sections on NVS. The segmented model with underlying paging is based on commonly used principles, as discussed in [De65, De70], and motivated by a number of real systems, especially Multics [Be72, Or72, MaDo74]. Readers familiar with the Multics virtual store may wish to only skim this section.

3.1. Introduction

Virtual storage provides the programmer with a logical view of memory, which is independent of the physical size and organisation. This frees the programmer from storage management issues such as program placement and relocation, fragmentation, and garbage collection.

The programmer within a source language views a *name space*, consisting of symbolic and relative numeric addresses. The symbolic name is resolved to a binary address by a *binding function*:

$$\beta : \Sigma^* \rightarrow V$$

In this function, $\langle v \rangle \in \Sigma^*$ is a symbolic representation of the address, and $v \in V$ is the binary *virtual address* to which it is bound. Symbolic addresses will be represented within angle brackets, following the convention used by Multics. When the binding is performed at compile, link, or load time, the binding is referred to as *static* (β_s). *Dynamic binding* (β_d) is performed on any unresolved symbols at execution time [MaDo74].

The binding function allows programs to use symbolic representations of addresses, but it is still desirable to allow the bound virtual addresses to be arbitrarily mapped to physical storage. The program virtual address is translated to a physical address at execution time by a *dynamic address translation function* (DAT):

$$\alpha : V \rightarrow R$$

where $v \in V$ is a virtual address, and $r \in R$ is a physical, or *real address.* If the mapping for a particular address does not exist ($\alpha(v) = \emptyset$), then the virtual address specified must be resolved to a location on auxiliary (backing) store:

$$\alpha' : V \rightarrow A$$

where $a \in A$ is the auxiliary storage location. The contents of a can then be moved to some $r \in R$ to create the desired mapping: $v\alpha r$.

If all processes in the system share the same address space V, a *single address space* (SAS) virtual store results. If each process (or set of processes spawned by a single parent), has a separate address space V_i, the virtual store is referred to as *multiple address space* (MAS). This means that each process i has its own DAT function

$$\alpha_i : V_i \rightarrow R$$

Particularly in a MAS system, the total virtual address space size may be much larger than the real address space. Therefore, assuming that the virtual storage is divided into blocks, only a subset of each process' blocks can be in real storage at a time, the *working set*. The working set can be modeled as those blocks $B_{\Delta t} \subseteq V$ that have been referenced during the interval $(t, t - \Delta t)$ [De68]. This is defined as the working set with window Δt. The working set of blocks in real store is an approximation of the process *locality set* [Ma76, Ma82], which is the subset of addresses in the address space referenced during a particular phase of the execution.

3.2. Linear address model

In the *linear address model*, a program (after static binding) is a sequence of addresses $V = \{0, 1, \ldots N\}$. The DAT function α maps these virtual addresses to a set of real addresses $R = \{0, 1, \ldots M\}$. Typically, $N > M$ to provide a virtual address space larger than the size of physical storage, but in some cases it makes sense for $N \leq M$ (*e.g.* in a time sharing system with many users).

The DAT function is performed by a table lookup. To avoid a table entry for every virtual address the address space is divided into blocks, allowing a single DAT table entry *per* block. For ease in storage management, the blocks may be of equal size, called *pages*. Thus, the low order bits of v are the offset into a page, and the high order bits define the page number, which is the index into a page table.

Arbitrary sharing of portions of address spaces between processes is difficult to manage, and can only be accomplished by agreement on the virtual (binary) address ahead of time. This may be done in systems by reserving virtual address ranges that are global to all processes, and by sharing entries in a second level page table [IBM83]. Dynamic binding and linking is also difficult to implement using a linear address model, since there are no logical address ranges to reference by name. Sharing at the page level imposes artificial boundaries, not related to program structure.

3.3. Segmented address model

In the segmented model, the process name space consists of a set of named segments, each of which is linearly addressed, either symbolically or by relative numeric offset. Note that the linear address model can be considered a special case of the segmented model, where there is only one segment. Segments are of variable size, where the length corresponds to the program or data object represented. Thus, the virtual address space consists of a set of segments $\{s_0, s_1, \ldots s_\ell\}$, with the address range of each segment $V_i = \{0, 1, \ldots N_i\}$.

Program addresses will be specified in the form $\langle s \rangle | o$ where $\langle s \rangle$ is the symbolic name of a segment, and o is a relative offset into the segment (statically bound to a numeric offset).

In contrast to the linear address model, all addresses corresponding to the binary representation of v do not exist, *i.e.* the address space is sparse. Only offsets bounded by the corresponding segment size are defined, and only segment identifiers corresponding to segments within the address space of the process are defined. This provides automatic checking of addressing within segment bounds. An invalid relative address (offset) still refers to the proper segment, and will generate an out-of-bounds addressing exception rather than referring to the wrong program or data object.

Sharing of segments between processes follows very easily, simply by providing accessibility to the same segments from multiple processes. The segment identifier ($v.s$) for a given segment need not be the same among processes sharing the segment, and thus no agreement needs to be made *a priori*. The segmented model provides a MAS virtual storage, and each process (set) has its own segment tables. The ability to share segments results in performance benefits. For example, at most one copy of a (pure, refreshable) code segment or read-only data segment need be active in the system at once, resulting in lower demands for real storage, allowing larger working sets and higher available CPU–memory bandwidth (due to less swapping of multiple copies of segments). Dynamic binding and linking also follows naturally, since symbolic addressing by segment name can be resolved to the appropriate segment table entry at execution time.

Shared segments provide an efficient and natural mechanism for IPC using the shared memory paradigm. Processes communicate through the use of shared variables in a data segment; the protection mechanism is automatic. In contrast, since in the linear addressing model the address range of shared space must be known *a priori*, shared addresses are typically global to all processes. The OS must manage access to this space, to insure protection and correctness between the various IPC process sets. In this case, much of the efficiency advantage is lost, and the implementation degenerates to that of message passing, where processes send and receive messages *via* system calls, which use the global shared space.

For performance reasons, a grouping of segments, called a *segment access group*, can be defined. This specifies a group of segments which are transferred within the storage hierarchy, whenever one of the segments is moved. This structure is also used to define the related segments for a sequential segment streaming operation. Note that this is similar to a higher level of segmentation, but its implementation does not involve part of the virtual address or address translation process.

3.4. Segmented model with underlying paging

Segmentation provides a model of virtual storage that is based on logical program structure, allows natural sharing of program and data objects, and provides the framework for dynamic linking and shared variable IPC. The benefits that paging provides to storage management can be extended to the segmented model, by allowing individual segments to be paged. Storage management is considerably easier in a paging system, since the placement policy is trivial, *i.e.* place a page in ANY free page frame. The complications of placement, free space management and compaction that are present under pure segmentation are eliminated. The process name space appears to be that of the segmented model. Segments are of variable size, but to the resolution of (the lowest level) page size.

The DAT function α maps the virtual address v to a real address r. The low order bits v are the offset into a segment ($v.o$), and the high order bits define the segment id., as in a pure segmented scheme. The low order bits of the segment offset are the offset into a page, and the high order bits of the segment offset define the page number within the segment. The DAT table lookup maps the segment id. and page number to a page frame in real storage. Thus, this results in a translation levels for both the segmentation and the underlying paging. In the simplest case, the virtual address is split into three fields:

$$v = v.s\ v.p\ v.o$$

The segment id. $v.s$ is the index into a segment table (whose base address is b_{ST}), which gives a pointer to the appropriate page table. The page number $v.p$ is the index into this page table, whose entry contains the address of the page frame in real storage. The translation function is

$$r = \alpha(v) = v.o + \alpha_{\text{PF}}(v.p + \alpha_{\text{PT}}(v.s + b_{\text{ST}}))$$

where α_{PF} returns the page frame address from the page table and α_{PT} returns the page table address from the segment table.

As described, the DAT function involves a multi-level table lookup, which results in significant overhead, since this must be performed for every instruction fetch and operand reference. This form of table translation will be referred to as α_t. An alternative, which involves very little overhead, is to use an associative map (α_a), which accesses all entries in parallel, within the instruction or operand fetch phase of the instruction cycle. The problem with this scheme is that the cost of the associative hardware is high, and grows significantly with the size of the associative table.

The typical approach is to combine the techniques, having a full set of translation tables, and a small associative map called a *translation lookaside buffer* (TLB), which contains recently used page frame address entries for each process, keyed on [*process-id.*, $v.s$, $v.p$]. Program address reference locality allows the number of entries to be small for each process, *e.g.* for a page size of $|p|$ a sequential instruction stream trace will hit the same TLB entry $|p|$ times. Sizes of only a few entries *per* process are quite adequate to give a high TLB hit ratio. This combination technique will be assumed to be in use from this point on.

Example implementation. A brief description of the structure of a possible implementation of a segmented virtual storage with underlying paging will be considered. The data structures described here are heavily influenced by Multics, as well as by NOS/VE and the System/38. A symbolic namespace is available to the programmer, with facilities for dynamic linking. The relationship of the tables discussed in this section are shown in Figure 2.

The system consists of a set of users. Each user u_j has accessibility to a set of segments U_j through a *user directory* UDir_j, which can be viewed as a (possibly structured) capability list. Processes are created by users, and the address space of a process x_i consists of a set of segments accessible through the UDir_j, $V_i \subseteq U_j$.

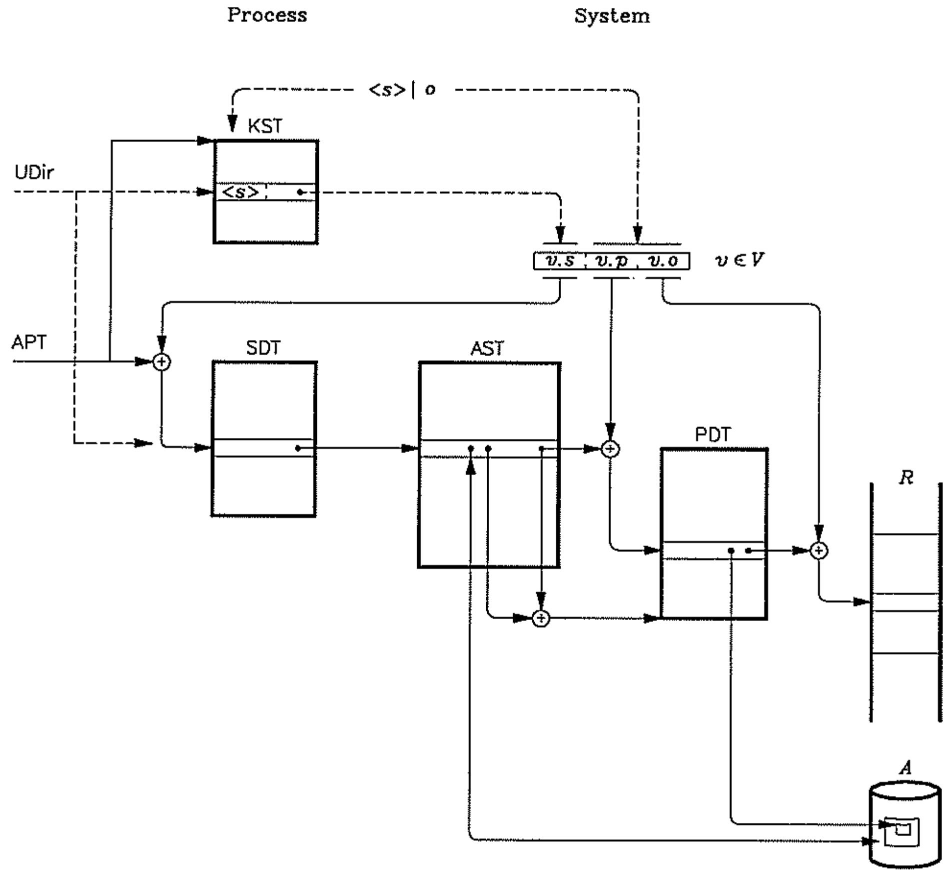


Figure 2: Segmented/Paged Address Translation

The address space of the process is made accessible through the *segment descriptor table* (SDT), which contains one entry (SDTE) *per* segment. When a process is invoked, the SDT is built to initially contain SDTEs for each segment which has been statically bound. If no static binding has taken place, the SDT initially contains an entry only for the executing code segment. If segment access groups are used, entries for all other segments in the access group of the executing segment will also be present. As the process executes, when an unbound symbol is referenced the segment is located through the UDir of the owner, and dynamically bound by adding an SDTE. To limit the overhead of UDir search to once for each different segment, a *known segment table* (KST) is built for the process, "cacheing" the bindings $\langle v\rangle\beta_d v$. Each time an unbound reference is encountered, the KST is searched. The UDir is searched only if the binding is absent from the KST, which is referred to as a *link fault*.

Segments are shared by the use of a system *active segment table* (AST), which contains the descriptors for every segment in the system in use by *some* process. A segment is shared when SDTEs from multiple processes point to a single ASTE (AST entry). Since the ASTE provides the common pointer to a segment, it also contains all descriptor information for the segment that is process independent (such as segment type, access allowed). Process specific descriptor information for each segment resides in the SDT (such as access granted to a particular process).

When a segment referenced is not present (*e.g.* for the first time after a link fault), referred to as a *segment fault*, the original copy is located on auxiliary storage. It is possible that the segment is in active use by another process. If this is the case the *activity bit* in the segment header will be

set, indicating that the *ASTE-pointer* field contains the address of the ASTE for the segment. In this case, the SDTE for the process segment-faulting is set to point to this ASTE.

The ASTE for each segment points to the corresponding *page descriptor table* (PDT) for the segment. Each PDT entry (PDTE) indicates if a page is in real storage with a *presence bit*, PDTE_p. If $\text{PDTE}_p = 1$, the mapping α is valid, and the *page frame address* field contains the real address of the page $(r.p)$. If $\text{PDTE}_p = \emptyset$, the auxiliary mapping α' must be used, and a page fault occurs, indicating that the operating system must retrieve the *page slot* from auxiliary storage using the *page slot location* field of the PDT $(a.p)$. Note that various organisations are possible for the page tables (*e.g.* hashed indices), but the implementation of page tables is not important to the extensions that will be described for the NVS environment.

4. Network Virtual Storage

This section describes the extensions to the basic segmented paged virtual storage model to allow segments to reside on non-local host systems. Then, the implementation of NVS in Axon is described, which is based on extension of the data structures and mechanisms of a Multics-like virtual store.

4.1. Address translation extended to the network

In the Axon environment, segments may be located in any system on the VHSI. Symbolic virtual addresses are a [*segment-name, offset*] pair or [*host-name, segment-name, offset*] triplet:

$$\langle v \rangle \mid o$$
$$\langle h \rangle \ \langle s \rangle \mid o$$

In the former case, the *user directory* (UDir) which owns the process will resolve the segment name $\langle v \rangle$ to a host name $\langle h \rangle$ and segment name $\langle s \rangle$ (on host h, which may be local). The symbolic segment name is resolved to a binary address by a *binding function*:

$$\beta : \Sigma^* \rightarrow V$$

In this function, $\langle s \rangle \in \Sigma^*$ is a symbolic representation of the address, and $v \in V$ is the binary *virtual address* to which it is bound. When the binding is performed at compile, link, or load time, the binding is referred to as *static* (β_s). *Dynamic binding* (β_d) is performed on any unresolved symbols at execution time [MaDo74].

The symbolic host name is bound to a network path or address, by a *host name resolution function*:

$$\nu : \Sigma^* \rightarrow H$$

where $\langle h \rangle \in \Sigma^*$ is the symbolic name for the host, and $h \in H$ is the network address or path (*e.g.* source route) of the host.

Note that as described here, the addition of a host portion of the virtual address does not change the size of the virtual address space, it merely indicates the distribution of segments throughout the network. Thus the basic unit of addressing to the program is still the segment. Under NVS, the DAT (dynamic address translation) function α_n maps the virtual address $v \in V$ to a real address $r \in R$ for a segment originating on a host $h \in H$. Once a copy of the segment has been obtained locally, the DAT function is the same as for a local segment:

$$\alpha_n : V \rightarrow R$$

but if $\alpha_n(v) = \emptyset$ the segment must be located on auxiliary (secondary) storage A on the appropriate host in the internetwork:

$$\alpha'_n : V \rightarrow H \times A$$

There are two ways to account for segments located on other hosts. In the first case, the virtual address is split into four fields, and referred to as a *network virtual address* (NVA):

$$v = v.h \; v.s \; v.p \; v.o$$

The host id. $v.h$ is the index into a host address table. The segment id. $v.s$ is the index into a segment table (whose base address is b_{ST}) and the page number $v.p$ is the index into the corresponding page table, giving the page frame address in real storage ($r.p$) and the page slot location in auxiliary storage ($a.p$). The translation functions are

$$h = \nu(v.h)$$
$$r = \alpha(v) = v.o + \alpha_{\text{PF}}(v.p + \alpha_{\text{PT}}(v.s + b_{\text{ST}}))$$

where α_{PF} returns the page frame address from the page table and α_{PT} returns the page table address from the segment table.

In the case of a non-local segment, once the host name is resolved, the transport protocol (ALTP-OT) is called to get a copy of the segment from host h. Location of $\langle s \rangle$ on h proceeds in the same manner as for an h-local segment-fault. ALTP-OT returns a copy of the segment to the local host, and the segment descriptors are built. Thus the processing of a segment fault is extended to include *remote link faults* which bind to host id. and may interact with an internet name server, and *remote segment faults* which require interaction between the virtual and real storage management routines and the transport protocol.

Since the address space of a process is defined as the set of accessible segments, and the host name is only necessary on a *per* segment basis, $v.h$ is not necessary in the virtual address for address translation. Thus, in this second case, $v.h$ is located in the the segment descriptor, and is not necessary for execution once a segment has been fetched to the local system. This allows the use of a *local virtual address* (LVA) by the DAT hardware, with $v.h$ extracted from the segment descriptor only on remote segment fault. It should be noted that LVA based translation has the obvious advantage of more compact address representation, as well as compatibility with many existing virtual address formats. There may some host architectures, however, where address space size is not a problem and the use of NVAs is desirable, since the presence of $v.h$ as part of the virtual address provides immediate access without indirection to the SDT (*e.g.* in the case of page faulting across the network – see ASR remote placement policy in §5.4), or in the case of a subnetwork of hosts that form a dense segment space with id. $[v.h \; v.s]$.

Multiple levels of host addresses, as well as multiple segmentation and paging levels, are possible, with virtual addresses of the form

$$v = v.h_q \; v.h_{q-1} \ldots v.h_1 \; v.s_n \; v.s_{n-1} \ldots v.s_1 \; v.p_m \; v.p_{m-1} \ldots v.p_1 \; v.o$$

This would correspond to a hierarchy of networks (*e.g.* local net, subnet, internet autonomous domain), containing a hierarchical organisation of segments, with multiple levels of paging tables underneath. It will be assumed for this discussion that any host hierarchy is collapsed into a single level of host name (h_1). In particular, if a hierarchical host name (such as in the DOD Internet domain name system) is used, the hierarchy will be transparent to the local system, and will be resolved by a hierarchical internet name server. A single level of segmentation above a single paging level will also be assumed.

Note that NVS might be thought of as simply another level in the address mapping hierarchy: cache → real storage → auxiliary storage → network. The behavior of 3-level addressing is well understood [Ma77]. A very important distinction exists for the network level, however. In the lower levels, strict assumptions can be made about the behavior of the system. In particular, the network level adds considerable complexity in the effect of errors (packet duplication, loss, and sequencing). In addition, at the network level the *distribution* of end-to-end latency may be considerably different from lower levels. Furthermore, issues concerning flow/rate control must be considered. It is the transport protocol (ALTP-OT) that hides this additional complexity from the programmer. Finally, with the incorporation of video segments the lower level mappings are different than for the case of code and data segments, *e.g.* the video frame buffer must be incorporated into the storage hierarchy.

4.2. Implementation

This section provides a brief description of the implementation of the Axon network virtual storage address translation data structures and mechanism. The relationship of the tables discussed in this section are shown in Figures 3 (NVA) and 4 (LVA). Address pointers and relative offsets are represented by arrows on solid lines, and other location information (*e.g.* disk *cylinder, track, record*) by arrows on lines with infrequent breaks. The copying of data structures (*e.g.* a segment) or fields (*e.g.* descriptor information) is represented by arrows on dashed lines. Note that the alternate paths for the returning segment s labeled RS and AS correspond to the remote placement policies *real storage* and *auxiliary storage* (§5.4).

Segment types. Axon segments are of two types: *memory* and *video*. Memory segments are either *code* or *data* subtype. Memory segments are divided into pages, and may be organised into segment access groups of related segments for performance reasons. Video segments are either *text* or *graphics* subtype. Graphics segments are bit-mapped video image frames; text segments correspond to a text window on a workstation. Video graphics segments are divided into scanlines, and may be organised into multi-plane images (*e.g.* a color image of R+G+B frames).

Segments have attributes of *read, write, execute*, indicating the type of access allowed. These access bits in the segment descriptor may differ from the (more restrictive) capabilities that individual users possess, or the descriptors of individual processes. Code segments are assumed to be pure (refreshable), and therefore always have access attributes of execute-only. Data segments may be readable and/or writable.

Host name binding. For NVS a table which maintains the host name to address resolution $\langle h \rangle \nu h$ is necessary, so that ν is performed only once *per* host *per* process. This is the *known host table* (KHT), and for each process contains fields for *host name* $\langle h \rangle$, *host address/path* (h) and *connection id.*

If the host name binding is not in the process' KHT, a *KHT-miss* occurs. In this case, the systemwide *host address table* (HAT) is searched which maintains the host name resolution binding ν_s for the entire local system, consisting of *host name, host address,* and *connection id.* fields as in the KHT. If the host address/path is not known to the entire system, a *host address fault* occurs, and a request is made to an internet name server. The *time-to-live* field indicates the life of the binding, after which the HAT entry (HATE) is deleted. The HATE also points to a chain of pointers to ASTEs (AST entries), indicating active segments to be located originating from each host. This allows location of a segment already locally active due to a remote segment fault by another process (assuming that global access is allowed; otherwise the request will have to be authenticated by the owning host).

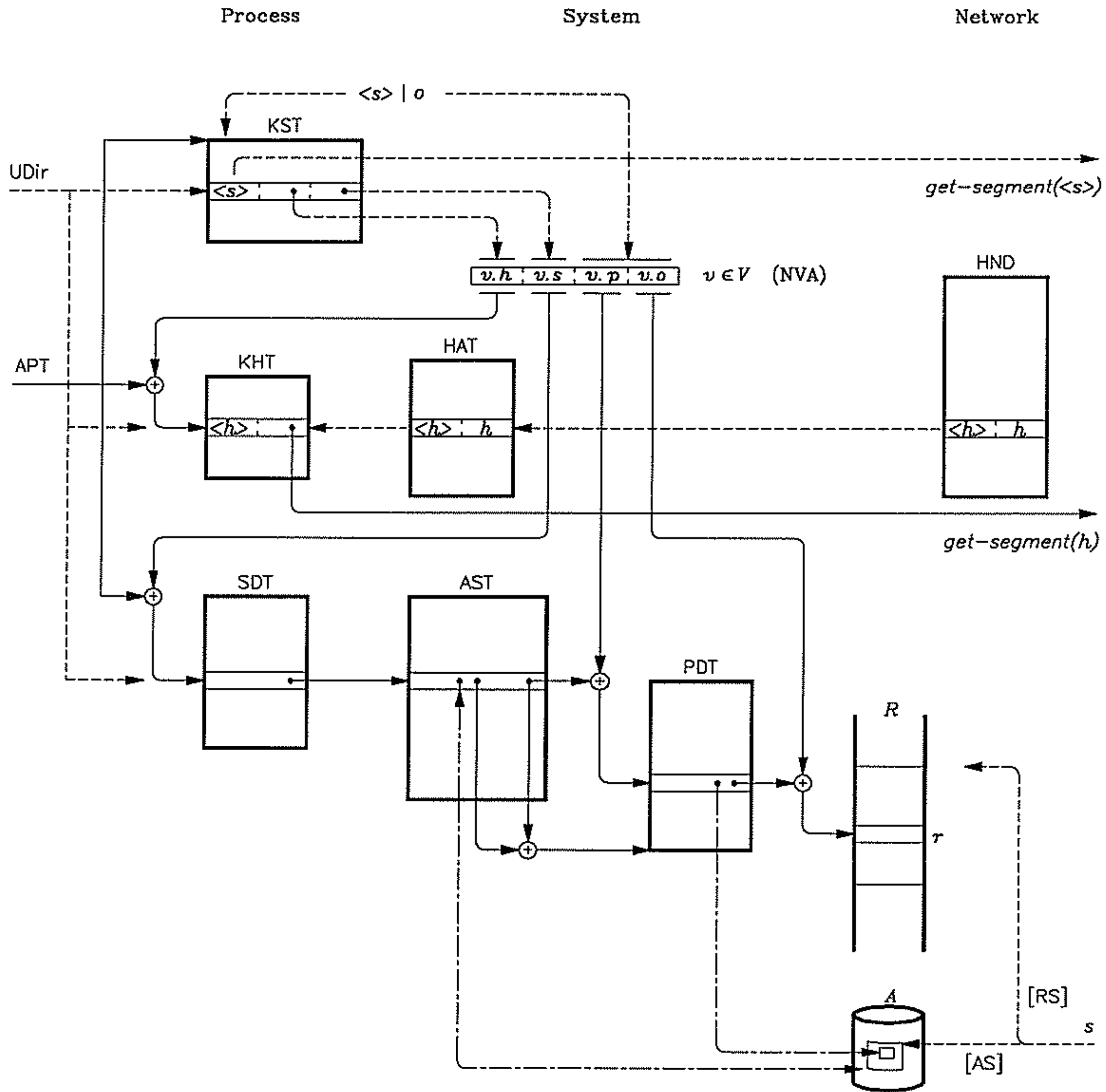


Figure 3: Network Virtual Storage Address Translation (NVA)

The *host name database* (HND) is a generic name for the (possibly distributed) database that the internet name servers use to perform the host name to address/path resolution ν_i, when requested by a host. This structure is assumed to exist, and be transparent to local hosts.

As in the local environment, the *known segment table* (KST) maintains the binding function $\langle s\rangle\beta_d v.s$, but a field must be added to indicate the host from which the segment came. Thus the KST entries contain the *host id.* ($v.h$) field along with with the *segment name* $\langle s\rangle$ and *segment id.* ($v.s$).

Segment descriptors. The *segment descriptor table* (SDT) contains descriptors for each segment in the process address space, consisting of the *process specific* attributes. The fields are access bits (*read, write, execute, shared*), state bits (*valid, remote-copy*), pointers to the segment's *active segment table* (AST) entry and *page descriptor table* (PDT), the *segment location* ($a.s$) in auxiliary

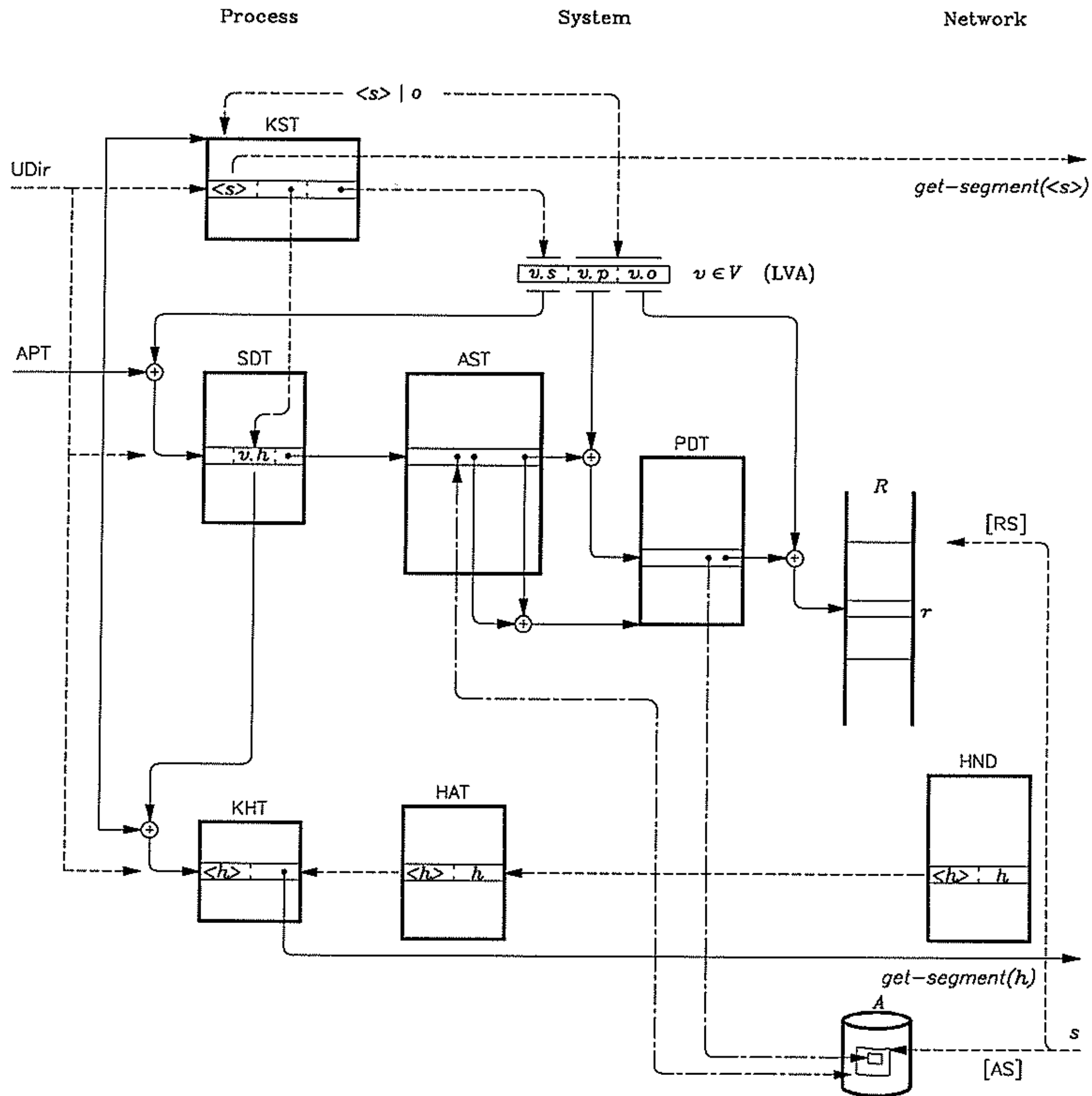


Figure 4: Network Virtual Storage Address Translation (LVA)

store, and the *host id.* for LVA format translation. Some of the SDT fields mirror AST fields, to allow efficient access without indirection to the AST.

The AST (active segment table), which contains the descriptors for every segment in the system, consists of the *segment specific* attributes. The AST entry has bits indicating the segment *type* and *subtype*, state bits (*present, valid, copy, swappable*), access rights (*read, write, execute*), cache control bits (*cacheable, write-through*), the *segment location* ($a.s$), and pointers to the PDT and owning process' APTE (active process table entry). NVS requires several additional fields not required in the local environment. The *copy* bit indicates if the segment is a copy from a remote host. The *valid* bit indicates if the segment contains valid data, or if it has been invalidated by the segment coherency protocol. The *locked priority* and *queue* fields are also used by the segment coherency protocol. The *connection id.* field associates a remote segment with a particular MCHIP internet level logical connection.

The PDT (page descriptor table) contains descriptors for each page in a segment, consisting of the *page specific* attributes. The fields are access bits (*read, write, execute, shared*), *page-presence*, state bits (*nailed, referenced, modified, [segment-]locked*), cache control bits (*cacheable, write-through*), an *unreferenced interval time* field for the replacement policy, real storage *page frame address*, and auxiliary storage *page slot location*. Note that since sharing is done at the segment level, NVS has little impact on the page table organisation, which may be whatever is appropriate for a given host system. The exception is that if the PDT is used to mark packet arrival, the presence field is supplemented by a *packet-vector*, which is a bit vector marking presence for each packet in the page.

Remote segment location. Three mechanisms allow segment location at the remote end of the connection (*e.g.* in response to a get-segment), as indicated in Figure 5.

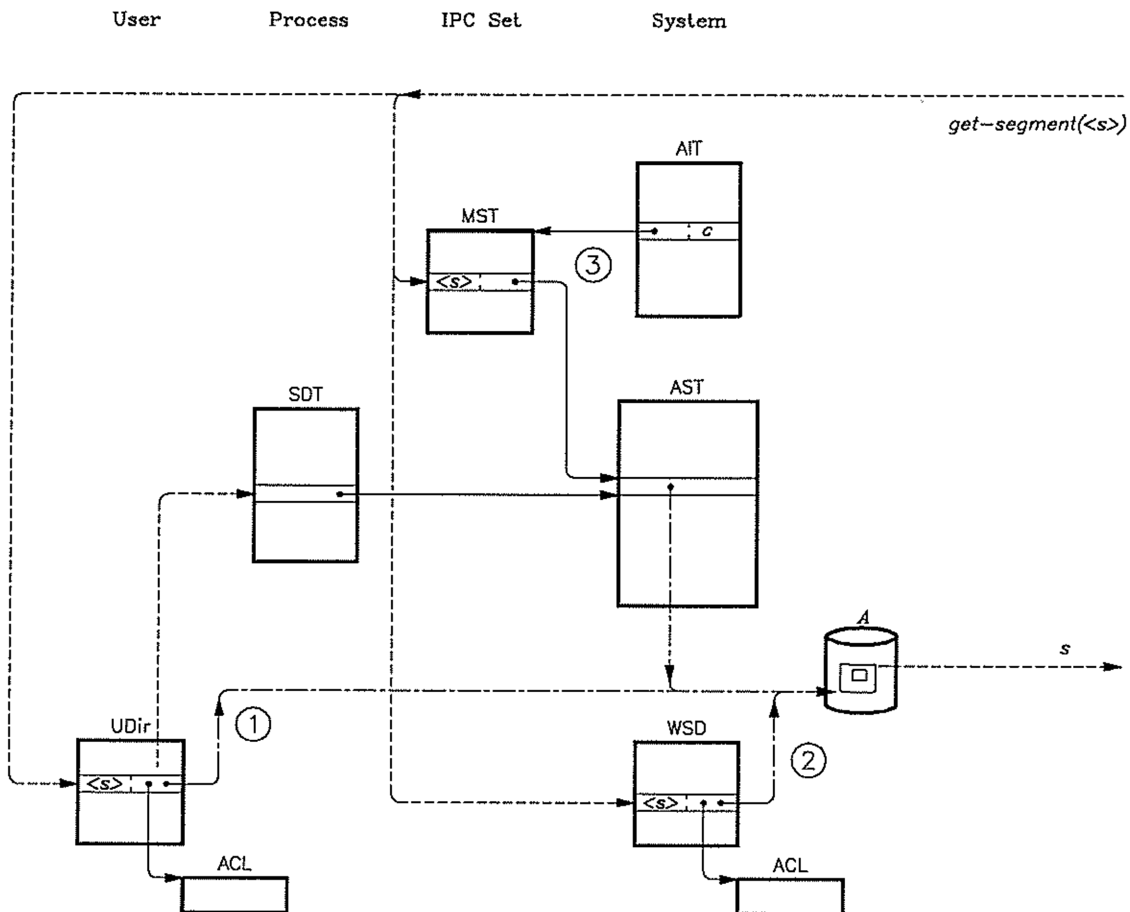


Figure 5: Remote Segment Location

[1] A segment request can be associated with a userid at the remote end. In this case, the associated UDir will be searched, just as for a local request. If the segment attributes allow public access, the segment can be returned to the local end. Otherwise, an *access control list* (ACL) associated with the segment will be checked with the userid and encrypted password of the requester to check authorisation.

[2] Segment requests can be for *well-known segments*, which have had their names advertised throughout the VHSI, and are found through the systemwide *well-known segment directory* (WSD).

Access to these segments can either be public, or with ACL verification, as in the previous case. The WSD entries contain the *segment name* ⟨*s*⟩, *segment location* (*a.s*), access rights (*read, write, execute*), and pointers to the segment *owner* and *ACL*.

[3] When process are engaged in a n-way IPC connection, special data structures assist in segment location, based on the connection id. (which corresponds to the MCHIP congram id. [MaPa89, Pa90]). Each IPC-set has its common portion of the address space defined by a *multipoint IPC segment table* (MST), which provides accessibility to all shared segments involved in the IPC by *connection id.* The MST entries have a segment name field ⟨*s*⟩, and pointers to the segment's ASTE and the owner's APTE. The system *active interprocess communication table* (AIT) locates (by connection id. and possibly IPC subset id.) the MST for a given IPC set. Note that all of the segments identified by the MST must be active (for some process), and therefore have descriptors in the owner's SDT and AST for each host involved in the n-way IPC.

Performance aspects. NVS in Axon is based on natural extensions to data structures and fault resolution mechanisms of a standard segmented paged virtual store, and many of the performance issues are similar. In particular, the overhead in DAT table lookup is much the same, with the use of a TLB to provide adequate performance. Thus the performance of NVS address translation is the same as for conventional systems, except for *remote* link and segment faults. The performance of a remote segment fault is primarily related to the end-to-end latency of the internetwork, the bandwidth of the connection, and the performance of ALTP-OT, and is thus not discussed here. The significant performance issues that remain are related to storage management policies, which are discussed in Section 5.

It should be noted that the speed-of-light component of end-to-end latency does not scale to VHSI data rates, and is increasingly dominant for wide area networks. One significant benefit of segmentation as the unit of object transport across the network is that program and data structure locality is exploited to to prefetch and cache pages, avoiding *per* page fault delay. Certain applications may also prefetch segments based on knowledge that they will be needed, either by reference or an explicit ALTP-OT get-segment call.

4.3. Example NVS segment transfer

The NVS mechanisms and relationship to ALTP-OT operation in the Axon architecture will be introduced by the description of a segment transfer. Explicit references to Figure 6 in this discussion are enclosed in brackets: ⟦ ⟧. Figures 3 and 4 can be consulted for the relationship between NVS data structures. Note that certain assumptions and policy choices have been made for clarity in this discussion.

An executing process has associated with it a virtual address space, which is a subset of the segments available to the user which owns the process. When a process refers to a remote segment, either explicitly by name, or *via* a GRPC, the appropriate segments must be transported from the desired system. The segment is located, either by an explicit reference to the segment and host name, or by resolution of the host name associated with the segment capability stored in the user context directory (UDir). The first time a segment is referred to symbolically, a *link fault* resolves the name and location, and adds the segment binding to the KST (known segment table), and host name binding to the KHT (known host table) for the process. This allows further *symbolic* references to avoid the overhead of searching the user context for segment attributes. In addition, an entry is added to the process SDT (segment descriptor table), which contains the process specific attributes of the segment. An entry is added to the system AST (active segment table), which contains the attributes of the segment common to all processes sharing the segment, if the segment is not already

Figure 6: Interaction of Axon Components

in use by another process. The mechanism for sharing is to have the SDT entries of multiple processes pointing to a single AST descriptor, which refers to a single instantiation of the segment. When a remote segment transfer is necessary, the transport mechanism is accomplished by ALTP-OT.

The critical path function of ALTP-OT is implemented in the CMP hardware ⟦ALTP-critical⟧, and includes the data path and routine control functions (error and flow control). The non-critical part resides in the systems software on the host (or CMP assist processor [St90]), and is tightly integrated with the host architecture and operating system ⟦ALTP-host⟧. In particular, the host portion of ALTP-OT must have direct access to operating system services such as the scheduler ⟦OS-sched⟧ through lightweight system calls, and be able to manipulate virtual storage management data structures ⟦VS-tables⟧.

The remote transfer is initiated by an ALTP-OT operation such as **get-segment**, which retrieves a segment from a remote host for local use. This requires a connection between the two hosts, thus ALTP-OT issues an **open** call to MCHIP which establishes the connection if not already present from a previous call. In addition, the CMP data pipeline is configured appropriately for the connection. ALTP-OT then sends the **get-segment** control packet out the VHSI link interface and through the internet, using the established connection.

At the remote end, the CMP receives and decodes the control packet at the internet link interface, and passes it to the host operating system. The normal mechanisms for locating the segment and authenticating the request are used. When the segment is found, locks are set (if necessary), and a copy of the segment is returned to the requesting host in a super-packet along the same connection. The data packets consist of fragments from each page of the segment, with an integral number of packets *per* page. Note that if multiple segments are defined within a segment access group, all of

them are returned in a single super-packet. Thus the unit of structure is a superpacket $[\![\sigma]\!]$ consisting of a segment group $[\![g]\!]$ of segments $[\![s]\!]$ of pages $[\![p]\!]$ of packets $[\![\pi]\!]$.

At the local end, storage has been allocated for the returning segment(s), based on the estimated segment size $|\hat{s}|$ and remote segment placement policy in use (either $[\![$real store frames$]\!]$ or $[\![$aux store slots$]\!]$). The data packets contain the actual segment size $|s|$, allowing adjustments to be made in the estimated storage allocation. The header of each data packet also indicates the packet and page (and segment) number (i, j, k), as well as the connection and request ids. Since the connection has been established, the CMP pipeline configured, and storage allocated, packets are placed directly in storage according to the remote placement policy; no buffering of the data by the CMP takes place, the order of packet arrival is not significant (*sequence by placement*), and there is no involvement of MCHIP or the host software portion of ALTP-OT. The structure of data between the CMP and target memory is the page $[\![p']\!]$. Note that the peer-to-peer connection between ALTPs is *physical*, without the strict calling and data copying involved in the OSI or other layered models, and there is none of the overhead associated with multiple packet encapsulation/decapsulation between layers.

When certain events occur, the CMP issues a signal to the host software portion of ALTP-OT. For example, each time all of the packets of a given page have been received, the presence bit in the PDT (page descriptor table) must be set, and a lightweight system call must indicate to the low level scheduler that the process can be dispatched, as in the standard page fault recovery mechanism. When the entire segment has been received, the presence bit in the AST (active segment table) is set, and the ALTP-OT connection idles until the process ends, or an explicit leave-ipc is issued.

5. Nvs Storage Management Strategies

Given the importance of storage management policies in determining virtual storage performance in the local environment, they must be reconsidered in Axon to allow the optimisation of NVS performance, as well as to insure that local operation is not adversely affected by remote segment access. These policies are *fetch* (when to get an auxiliary storage object for real store), *placement* (where in real storage to put it), and *replacement* (which object to replace to auxiliary storage to make space for new fetches). In the NVS environment an entirely new storage management policy arises: that of *remote segment placement*, *i.e.* where segments are physically placed that become part of the local virtual address space when copied across the VHSI. Each of these policies will be discussed in the context of NVS, with particular emphasis on the replacement and remote placement policies.

5.1. Fetch policy

Local to a system, the fetch policy is that of *demand paging*. Under NVS, segments that are on a remote host are fetched *demand segment*, which can be thought of as *anticipatory* at the page level, since when a segment is fetched across the internet all of the segment's pages are fetched in anticipation of their possible use. This takes advantage of the program and data structure manifest in the segmentation to help overcome the end-to-end latency effects in a wide area network or internetwork.

5.2. Real placement policy

The placement policy determines where objects are placed in real storage. In a local environment, the consideration is where objects will be placed in real storage to maximise storage utilisation and minimise overhead. In the NVS environment this will be referred to as the *real placement policy*.

NVS has no impact on this policy since in a paged system all units of data transferred between real and auxiliary store are of uniform size, and the placement policy is trivial: use ANY free page frame.

5.3. Replacement policy

There are two levels at which replacement policies are typically implemented. Hardware replacement can be implemented as *not-used-recently* (NUR), which uses bits indicating that a page has been accessed or modified to determine a reasonable candidate for replacement. NUR gives a reasonable approximation to the performance of one of the more sophisticated algorithms, such as *least-recently-used* (LRU), but with an efficient hardware implementation. In Axon, host systems use whatever hardware replacement policy is appropriate, such as NUR.

At the software level, the operating system should attempt to maintain a pool of free pages, avoiding the need to replace a page using hardware replacement. Thus when space for a page is needed, it is simply reclaimed from the free pool, rather than possibly requiring that a dirty page be copied back to auxiliary storage. The software algorithm can be any reasonable algorithm, such as LRU, which provides a good estimate of the locality set. This can run concurrently with all other processes, and use idle CPU cycles and I/O bandwidth to maintain the desired free pool.

Assume that the working set of pages in real storage is $P_{\Delta t} \subseteq V$ (working set window of Δt) [De68]. In the NVS environment, pages from local segments P_ℓ are considered separately from those from remote segments P_r, the working set is defined as $P_{\ell \Delta t_\ell} \cup P_{r \Delta t_r} = P_{\Delta t}$. By setting $t_r > t_\ell$, remote segments are given a relatively larger share of the process working sets, and are therefore subject to less page fault delay. For processes that use a remote segment for a short period of time, this may reduce the total time that a remote segment is held and possibly locked from other processes. An alternative is to increase the entire (remote *and local*) working set of processes with *any* remote segments, over processes that have *no* remote segments. This avoids increasing the page fault delay for local segments of processes having some remote segments.

The implementation of this extended working set algorithm is to use LRU, with virtual time $t_r - t_\ell$ subtracted from the unreferenced interval time field for the appropriate pages. This gives the appearance that these pages have been more recently used. This can be done either to all remote pages, or to all pages for processes that are using any remote segments, as discussed above. Another mechanism for favoring processes which hold remote segments is to increase their dispatching priority. The processes with remote segments can also be favored by marking them non-swappable ($\text{ASTE}_s = \emptyset$).

The effectiveness of these schemes is somewhat dependent upon the segment coherency algorithm. A detailed exploration of coherency algorithms is beyond the scope of this paper; a simple priority based segment locking scheme is currently assumed.

5.4. Remote segment placement policy

In Axon NVS, the virtual address space $V = V_\ell \cup V_r$ of a process consists of a set of local segments V_ℓ and remote segments V_r. Since segments of variable size are fetched across the VHSI, there is significant potential for performance impact and the placement of the segment becomes important. The algorithm for determining the placement of remote segments is called the *remote segment placement policy*. A number of remote segment placement policies can be identified:

AS – auxiliary (or extended) storage placement:
Two possibilities exist for the placement of segments in auxiliary storage. Segments can be placed in *local* auxiliary storage (ASL), or remain in remote auxiliary storage (ASR).

ASL – auxiliary storage local: Remote segments V_r are fetched across the VHSI, and placed in local auxiliary storage, along with all of the active copies of local segments V_ℓ. Pages are moved to real storage when a page fault occurs, in exactly the same manner as for local segments. The sub-policy for placement within auxiliary storage needs to be considered, especially if contiguous disk allocation is used (*first-fit, best-fit, etc*). In the case of a video segment, the normal mechanism for fetching to the frame buffer is used. This is the simplest scheme to implement from the viewpoint of the host operating system.

ASR – auxiliary storage remote: Remote segments are left on the auxiliary storage of the remote system. In this case, only *pages* are fetched across the network, with each page fault. This policy makes sense only when the network latency is considerably less than the disk latency, *e.g.* in a LAN subnet. On the other hand, in a high performance local environment, active pages would be located in extended storage or in the disk controller cache, whose access time may be below the latency of even a single LAN hop. Therefore the tradeoffs between ASL and ASR are directly affected by internet/subnet and real/auxiliary storage latencies.

RS – real storage placement:
In this scheme, segments are placed directly into page frames in real storage. The main disadvantage of RS is that this causes the working set of processes to contain *all* pages of remote segments, including those which are not part of the locality set for each execution phase. Clearly this could cause a shortage of real storage for pages of local processes. On the other hand, Axon provides a direct path from CMP to real storage operating at VHSI network bandwidths. The path to auxiliary storage may be much slower, introducing considerable blocking if AS is used, particularly in the absence of extended memory or auxiliary storage controller cache. The choice of RS *vs.* AS is also affected by the host architectural configuration (§2.4). In particular, memory interface architecture strongly suggests RS, since the CMP is directly connected to a CMM which is part of real memory. Variants of RS are possible, based on the pagability and swapability of the segment:

RSN – real storage nailed: the page frames are *nailed* (non-pagable: $\text{PDTE}_n = \emptyset$) in real storage, *i.e.* the page frames are marked non-pagable, and the segment is non-swappable ($\text{ASTE}_s = \emptyset$). This is an attempt to improve performance for processes using remote segments, allowing them to be released more quickly.

RSS – real storage swappable/nailed: the page frames are nailed (*i.e.* no individual page movement takes place) but the segment may be *swapped* to auxiliary store (allowing the OS to adjust multiprogramming level and real storage availability). This requires that the swapping routines called by the intermediate scheduler recognise that a copy does not exist in auxiliary storage, and swap out the entire segment rather than just the working set, and is indicated by a remote segment ($\text{ASTE}_c = 1$) that is swappable ($\text{ASTE}_s = 1$) but not pagable ($\text{PDTE}_n = \emptyset$). RSS provides some of the benefits of RSN in avoiding the delay associated with page faults, while still allowing more performance and load balancing control to be exercised by the OS as in ASL.

RAS – real and auxiliary (or extended) storage placement:
In this scheme an initial working set estimate of the pages $\hat{P}_{\Delta t} \subseteq V_r$ is placed in real storage, and all segments V_r are backed to local auxiliary store. Paging and (and conditionally) swapping are allowed as for any local segment. Clearly the first page referenced should be one of the pages placed in real storage. This page is determined by ALTP-OT based on the offset portion of the virtual address. Any additional pages placed are based on a prediction of the memory reference trace $\rho_{(t,t+\Delta t)}$. It is not clear that this scheme would be any more successful than an anticipatory paging scheme in the local environment, although if the operating system performs limited page prefetches (such as the NEXT sequential page), this may be applied in a like manner for RAS.

FB – frame buffer placement:
This policy corresponds to RS, but for video segments (assuming that the frame buffer is not real-memory mapped). This requires that packets are reorganised into the frame buffer format in real time, including the spatial transformation corresponding to the appropriate coordinates of the target window. FB may also have implications on error control exercised by ALTP, in particular relating to packet sequence. One solution is to utilise a double buffered structure, with the monitor driven from the active frame buffer, and a shadow frame buffer which is loaded from the CMP. When an entire segment is present in the shadow buffer, the window can be moved to the active frame buffer. In addition, the structure of the shadow buffer should account for windows which are obscured by other windows, and hold the entire contents of all windows for quick response of workstation window manipulation.

6. Related Work

Early work in the research community on IPC and the design of distributed systems was done in the context of tightly coupled multiprocessor systems, as opposed to loosely coupled systems situated across local and wide area networks. There were only a few exceptions to this trend, including DCS [Fa88, Fa75, FaFe73, FaHe70], and the Newhall ring [MaPe75], but these both were in the LAN context.

There has been some research on exploring the shared memory paradigm for IPC over the network, exemplified by Memnet and LOCUS. LOCUS [PoWa85, WaPo83, PoWa81], a UNIX variant based on a distributed file system, has had UNIX System V communication primitives added, specifically, *messages*, *semaphores*, and *shared-memory* [Fl86], with current work extending this support to provide the *shared-memory* across a network [Fl87].

In the case of Memnet [De88, DeSe88, DeFa85], processes communicate across a ring LAN by reading and writing into shared memory. Memnet's emphasis has been on studying cacheing algorithms and their hardware implementations, to reduce the network traffic and to avoid network latency for remote memory accesses. However, Memnet assumes a perfect communication medium with no errors, and does not allow virtual storage. The CapNet project [TaFa89] is extending the Memnet work in directions complementary to Axon, but with different emphasis. The Apollo DOMAIN [LeLe83, Ap85] system also provides a shared memory interface on a LAN ring.

There are also other research groups that are starting to explore use of shared memory for IPC across network, including current work on Ivy, Mermaid, Shiva, Ra, and the Tapestry project [CaRe88, CaRu88]. The Ivy [Li86, LiHu90], Mermaid [LiSt88], and Shiva [LiSc89] research explores a shared virtual memory, with particular emphasis on providing page level coherency, and accommodating heterogeneous systems. Unlike Axon, the granularity of object transfer is the page, rather than the segment. The Ra [AuHu87] kernel project for the Clouds distributed system includes an investigation of distributed shared memory (DSM). This consists of exploring alternative address translation schemes and memory management hardware [RaKh88b], with particular emphasis on the object orientation of the system [RaKh88a].

A segmented, paged virtual store was first implemented by Multics [Be72, Or72, SpOr75] on a GE-645 and the IBM 360/67 [IBM72] running TSS/360 [Co65, Le65, IBM78a]. The Multics line continued on the HIS 645, 6180, DPS-60/68, DPS-8/M, but has now been terminated. Modern systems that owe significant heritage to Multics include the CDC Cyber 800 NOS/VE [CDC84a, CDC84b] and Prime 50 Primos [AuLa83a, AuLa83b]. Segmented virtual store was not used by other operating systems in the IBM System/360 and 370 family, until the addition of features provided by ESA/370 [ScGa89, Pl89, IBM88a] under MVS/ESA [IBM88b].

Additionally, systems that provide a segmented paged virtual store include the IBM AS/400 [IBM88c] and System/38 CPF [IBM78b, IBM86, IBM85], AT&T 3B series [HeKu83, ATT86a, ATT86b], Intel iAPX432 [In81, In83a, In83b] 80386 [In86b, In87] i486 [In89a, In89b] and 80960 [In88], and Motorola 68030 [Mo87].

7. Conclusions

There has been significant progress in the areas of communication and computer architecture over the past few years. We will soon have communication networks and internetworks that can support data rates of more than hundreds of Mbps, and have computers that can process numerous demanding applications such as video distribution, computer imaging, distributed scientific computation and visualization, distributed file and procedure access, and multimedia conferencing. These applications in a network environment can be characterised as needing to transmit large bursts of data with sub-second latency. The major bottleneck in supporting these applications remains at the host–network interface. In particular, IPC performance across a network has not kept pace with the demands of applications and the data rates of underlying networks.

We have proposed a new host communication architecture for distributed systems called Axon, which can support IPC with high throughput and low latency across the VHSI. In this paper we have described the design of NVS in the Axon architecture. This includes the extension of the standard segmented paged virtual storage model to include segments across the VHSI, and discussion of various storage management policies and their tradeoffs in the NVS environment. Work is in progress on analytical and simulation models to evaluate these tradeoffs more rigorously, and on a prototype Axon implementation.

A. Network Virtual Storage Data Structures

This appendix gives the specification for the data structures used by NVS. For each table, the existence item indicates whether it is present on a *per* user, process, system, or internetwork basis. Then, a brief summary of the usage of each table is given, followed by an indication of what part of the table supports the remote aspects of the NVS environment, which would not be required in a conventional tightly coupled virtual storage system.

Finally, the format of each structure is given. Pointers are indicated by the $\uparrow$ symbol, *e.g.* $\uparrow x$ is the address of x. A location indication of an object, which is not an address, is indicated by the $\Uparrow$ symbol, *e.g.* $\Uparrow s$ indicates the location of s, which might be in the form of disk [*cylinder, track, sector*]. At the top of each table, the descriptive name of each field is given. Within each table, a row is shown, which gives the symbolic representation of the field (when appropriate). Fields which are bit strings are represented by a string of variables; a descriptive name of each is below the field. The fields that index into a table are indicated next to the symbol $\downarrow$.

A.1. KST (Known Segment Table)

- Existence: *per* process
- Usage: Stores symbolic segment name bindings ($\beta_d : \langle s \rangle \mapsto v.s, v.h$) so that linkage fault resolution occurs only once per process per segment. The hostname $\langle h \rangle$ is maintained to allow the use of multiple segments from different hosts with the same name.
- NVS support: Host name and number fields $\langle h \rangle, (v.h)$
- Format:

name	segid	host	hostid	
$\langle s \rangle$	$v.s$	$\langle h \rangle$	$v.h$	KSTE
β_d $\longrightarrow$			$0 \equiv$ local	

A.2. KHT (Known Host Table)

- Existence: *per* process
- Usage: Stores symbolic host name to network address or path bindings ($\nu_p : \langle h \rangle \mapsto h$) so that host fault resolution occurs only once per process, and that the host name binding is independent of the binding life of HAT entries. The KHT is indexed by the host number portion of the virtual address ($v.h$).
- NVS support: Entire table
- Format:

	host	adr/path	connid	
↓				
$v.h$	$\langle h \rangle$	h	c	KHTE
	ν_p ⟶			

A.3. SDT (Segment Descriptor Table)

- Existence: *per* process
- Usage: Descriptors to provide accessibility to all segments/frames within the address space of the process. Contains segment attributes that are unique to the process. Address translation uses the SDT (*via* the ASTE) to locate the page table (PDT) for each segment. The SDT is indexed by the segment number portion of the virtual address ($v.s$).
- NVS support: Remote field; If LVA format address translation is used, a $v.h$ field is present.
- Format:

	↑ASTE	access	state	hostid	⇑seg	↑PDT	
↓							
$v.s$	e_a	$rwxs$	vc	$v.h$	$a.s$	b_p	SDTE
		read *write* *exec* *shared*	*valid* *remote*	LVA only	for no indirection		

A.4. AST (Active Segment Table)

- Existence: *per* system
- Usage: Descriptors to provide accessibility to all segments/frames in active use by the system. Contains segment attributes that are common to all processes sharing each segment. Provides segment accessibility for the system, and to the processes using each segment. The *system address space* is defined by the AST.
- NVS support: Copy (of remote segment) and valid state bits, connection id. field, and lock fields for segment coherency protocol; If the LVA format address is used, a $v.h$ field must also be present.
- Format:

$\Uparrow$seg	$\uparrow$PDT	length	type	subtype	state	cache	rights	lock	$\uparrow$APTE	cid	
$a.s$	b_p	l	m/w	$c/d \vee t/g$	$pvcs$	cw	rwx	$l\ p\ \uparrow$Q	e_t	c	ASTE
		$\lvert s \rvert / \lvert p \rvert$	*mem* *video*	*code* *data* *text* *graphics*	*pres* *valid* *copy* *swap*	*cache* *w-t*	*read* *write* *exec*	*locked* *prty* *queue*			

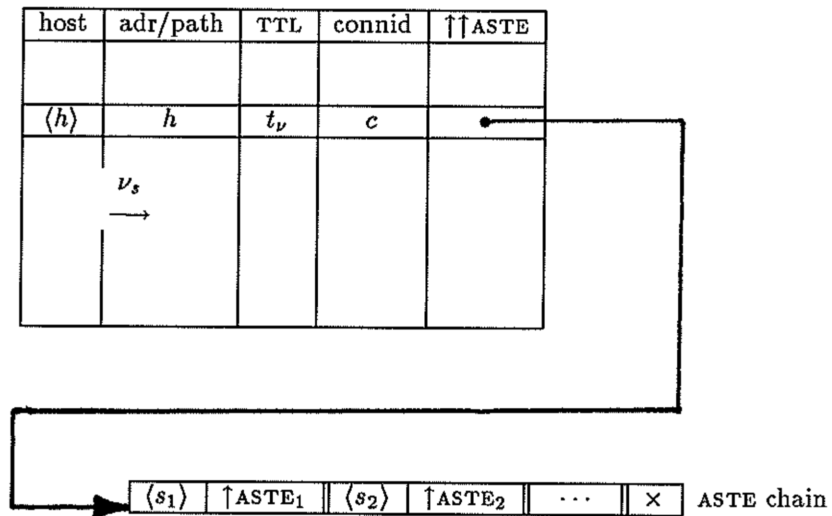
A.5. PDT (Page Descriptor Table)

- Existence: *per* segment, or a single system page table containing entries for all processes (may be preferred if a hash function is used to locate PDTEs).
- Usage: Locates pages/scanlines in real memory (*page frames*), and in auxiliary or extended memory (*page slots*), and provides state information for virtual storage management. The PDT is indexed by the page number bits of the offset portion of the virtual address ($v.p$).
- NVS support: Packet presence vector π if implicit PDT packet-to-page mapping is used. Host id. if LVA format address translation used with ASR remote placement policy.
- Format:

	$\uparrow$frame	$\Uparrow$slot	state	presence	cache	rights	uit	hostid	
$\downarrow$									
$v.p$	$r.p$	$a.p$	$nrml$	$p\pi_0 \ldots \pi_{\lvert p \rvert / \lvert \pi \rvert}$	cw	rwx	Δt	$v.h$	PDTE
			nailed *referenced* *modified* *locked*	*page-presence* *packet-vector*	*cache* *write-thru*	*read* *write* *exec*	unref intvl time		

A.6. HAT (Host Address Table)

- Existence: *per* system
- Usage: Caches symbolic host name to network address or path bindings ($\nu_s : \langle h \rangle \mapsto h$) for the entire system so that processes do not need to access a name server with every host fault. Provides accessibility to remote segments active from each host.
- NVS support: ASTE chain pointer and connection id. fields
- Format:



A.7. HND (Host Name Database)

- Existence: *per* internetwork – distributed among name servers
- Usage: Used by nameservers to resolve host names to network addresses or paths ($\nu_p : \langle h \rangle \mapsto h$)
- NVS support: None

A.8. WSD (Well-known Segment Directory)

- Existence: *per* system
- Usage: Descriptors providing access to segments whose name is advertised as *well-known* on a particular system.
- NVS support: Entire Table
- Format:

name	type	$\Uparrow$seg/WSD	rights	$\uparrow$parent	$\uparrow$owner	$\uparrow$ACL	
$\langle s \rangle$	s/d	$a.s$	rwx	b_w	e_u	l_c	WSDE
	segment directory		*read write exec*				

A.9. AIT (Active Interprocess Communication Table)

- Existence: *per* system
- Usage: Identifies n-way IPC sets based on connection id. and IPC subset id., and provides access to the associated segments and processes through the MST.
- NVS support: Entire Table
- Format:

connid	IPC subset	$\uparrow$MST	
c	i	b_m	AITE

A.10. MST (Multipoint Interprocess Communication Segment Table)

- Existence: *per* IPC set
- Usage: Identifies segments involved in an n-way IPC connection, allowing segment location by other hosts involved in the connection. If the segment is a code segment that has been instantiated as a process, a pointer is provided to the corresponding *active process table entry.*
- NVS support: Entire Table
- Format:

name	↑ASTE	↑APTE	
$\langle s \rangle$	e_a	e_t	MSTE

A.11. TLB (Translation Lookaside Buffer)

- Existence: *per* system
- Usage: Associative store of address mapping function ($\alpha_a : pid, v.s, v.p \mapsto r.p$) for fast access avoiding the overhead of table lookup. Separate entries are maintained for each running process.
- NVS support: Host id. for dense $[v.h, v.s]$ address space only
- Format:

processid	hostid	segid	page#	↑frame	state	cache	rights	
x	$v.h$	$v.s$	$v.p$	$r.p$	$vrmls$	cw	rwx	TLBE
			$\xrightarrow{\alpha_a}$		*valid* *referenced* *modified* *locked* *shared*	*cache* *write-thru*	*read* *write* *exec*	

References

[Ap85] *Programming with System Calls for Interprocess Communication*, Apollo Computer, Inc., Chelmsford, Mass., 005696, rev 00, 1985.

[ATT86a] *WE 32100 Microprocessor Information Manual*, AT&T Technologies, Allentown, Penn., 307-730 issue 2, 1986.

[ATT86b] *WE 32101 Memory Management Unit Information Manual*, AT&T Technologies, Allentown, Penn., 307-731, 1986.

[AuHu87] Aubán, José M. Bernabéu, Phillip W. Hutto, and M. Yousef Amin Khalidi, *The Architecture of the Ra Kernel*, Georgia Institute of Technology, School of Information and Computer Sciences, technical report GIT-ICS-87/35, Atlanta, 1987.

[AuLa83a] August, Martha and Sarah Lamb, *PRIME 50 Series Technical Summary*, Prime Corporation, Framingham, Mass., rev 19.1, DOC6904-191, 1st ed., 1983.

[AuLa83b] August, Martha, Alice Landy, and Marilyn Hammond, *PRIME System Architecture Reference Guide*, Prime Corporation, Framingham, Mass., rev 19.2, DOC3060-192P, 3rd ed., 1983.

[Be72] Bensoussan, A., C.T. Clingen, and R.C. Daley, "The Multics Virtual Memory: Concepts and Design", *Communications of the ACM*, Vol.15 #5, ACM, New York, May 1972, pp. 308–318.

[BhSt88] Bhatia, Anil, James P.G. Sterbenz, and Gurudatta M. Parulkar, *Comments on Proposed Transport Protocols*, Washington University Department of Computer Science, technical report WUCS-88-30, St. Louis, Oct. 1988.

[BiNe84] Birrell, A. and B. Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, Vol.2 #1, ACM, New York, Feb. 1984, pp. 39–59.

[CaRe88] Campbell, Roy H. and Daniel A. Reed, *Unifying Shared and Distributed Memory Parallel Systems*, University of Illinois Department of Computer Science, UIUCDCS-R-88-1449, Urbana, Illinois, Aug. 1988.

[CaRu88] Campbell, Roy, Vincent Russo, and Gary Johnston, *A Class Hierarchical, Object-Oriented Approach to Virtual Memory Management in Multiprocessor Operating Systems*, University of Illinois Department of Computer Science, UIUCDCS-R-88-1459, Urbana, Illinois, Sept. 1988.

[CDC84a] *System Architecture: Cyber 180 Systems*, Control Data Corporation, Minneapolis, 204 137, 1984.

[CDC84b] *CDC Cyber 170 and Cyber 180 Volume I: Virtual State System Description, Functional Descriptions*, Control Data Corporation, Minneapolis, 60462090, 1984.

[Co65] Comfort, Webb T., "A Computing System Design for User Service", *Proceedings of the Fall Joint Computer Conference*, Vol.27 Part I, AFIPS, Spartan Books, Washington D.C., 1965, pp. 619–626.

[De65] Dennis, J.B., "Segmentation and the Design of Multiprogrammed Systems", *Journal of the ACM*, Vol.12 #4, ACM, New York, Oct. 1965, pp. 589–602.

[De68] Denning, P.J., "The Working Set Model of for Program Behavior", *Communications of the ACM*, Vol.11 #5, ACM, New York, May 1968, pp. 323–333.

[De70] Denning, P.J., "Virtual Memory", *ACM Computing Surveys*, Vol.2 #3, ACM, New York, Sept. 1970, pp. 153–189.

[De88] Delp, Gary S., *The Architecture and Implementation of Memnet: A High-Speed Shared-Memory Computer Communication Network*, University of Delaware Department of Electrical Engineering, #88-05-1, Newark, Delaware, May 1988.

[DeFa85] Delp, Gary S. and David J. Farber, *Memnet: An Experiment in High Speed Memory Mapped Local Network Interfaces*, University of Delaware Department of Electrical Engineering, technical report #85-11-1, Newark, Del., Nov. 1985.

[DeSe88] Delp, Gary S., Adarshpal S. Sethi, and David J. Farber, "An Analysis of Memnet: An Experiment in High-Speed Shared-Memory Local Networking", *SIGCOMM '88 Symposium: Communications Architectures and Protocols (Computer Communication Review)*, Vol.18 #4, ACM SIGCOMM, New York, 1988, pp. 165–174.

[Fa75] Farber, David J., "A Ring Network", *Datamation*, Feb. 1975, pp. 44–46.

[Fa88] Farber, David J., "Some Thoughts on the Impact of Ultra-High-Speed Networking on Processor Interfaces", Univ. of Penn. Distributed Systems Laboratory unpublished note, April, 1988.

[FaFe73] Farber, David J., Julian Feldman, Frank R. Heinrich, Marsha D. Hopwood, Keneth C. Larson, Donald C. Loomis, and Lawrence A. Rowe. "The Distributed Computing System", *COMPCON 73 Digest of Papers: Computing Networks from Minis to Maxis – Are They for Real?*, IEEE Computer Society, New York, 1973, pp. 31–34.

[FaHe70] Farber, David J. and Frank R. Heinrich, "The Structure of a Distributed Computer System – The Distributed File System", *First International Conference on Computer Communication*, Washington, D.C., 1970, pp. 364–370.

[Fl86] Fleisch, Brett D., "Distributed System V IPC in LOCUS: A Design and Implementation Retrospective", *SIGCOMM '86 Symposium: Communications Architectures and Protocols (Computer Communication Review)*, Vol.16 #3, ACM SIGCOMM, New York, 1986, pp. 386–396.

[Fl87] Fleisch, Brett D., "Distributed Shared Memory in a Loosely Coupled Distributed System", *SIGCOMM '87 Symposium: Frontiers in Computer Communications Technology (Computer Communication Review)*, Vol.17 #5, ACM SIGCOMM, New York, 1987, pp. 317–327.

[HeKu83] Hetherington, I.K. and P. Kusulas, "3B20D Processor Memory Systems", *Bell System Technical Journal*, Vol.62 #1 Part 2, AT&T Co., New York, 1983, pp. 207–220.

[IBM72] *IBM System/360 Model 67 Functional Characteristics*, IBM Corporation, Poughkeepsie, New York, GA27-2719-2, 1972.

[IBM78a] *IBM Time Sharing System Concepts and Facilities*, IBM Corporation, Poughkeepsie, New York, GC28-2003-6, 1978.

[IBM78b] *IBM System/38 Technical Developments*, IBM Corporation, Rochester, Minn., G580-0237, 1978.

[IBM83] *MVS/Extended Architecture System Programming Library: Initialization and Tuning*, IBM Corporation, Poughkeepsie, New York, GC28-1149-1, 1983.

[IBM85] *IBM System/38 Functional Reference Manual, Volume 1*, IBM Corporation, Rochester, Minn., GA21-3331-6, 1985.

[IBM86] *IBM System/38 Functional Concepts Manual*, IBM Corporation, Rochester, Minn., GA21-9330-6, 1986.

[IBM88a] *IBM System/370 Enterprise System Architecture Principles of Operation*, IBM Corporation, Poughkeepsie, New York, SA22-7022-0, 1988.

[IBM88b] *MVS/Enterprise System Architecture System Programming Library: Application Development – Extended Addressability*, IBM Corporation, Poughkeepsie, New York, GC28-1854-0, 1988.

[IBM88c] *IBM Application System/400 Technology*, IBM Corporation, Rochester, Minn., SA21-9540-0, 1988.

[In81] *Introduction to the iAPX 432 Architecture*, Intel Corporation, Santa Clara, Calif., 171821-001, 1981, reprinted in: *Tutorial on Advanced Microprocessors and High-Level Language Computer Architecture*, Veljko Milutinović (ed.), IEEE Computer Society Press, Washington, D.C., 1986, pp. 358–421.

[In83a] *iAPX 432 General Data Processor Architecture Reference Manual*, Intel Corporation, Santa Clara, Calif., 171860-004, 1983.

[In83b] *iMAX 432 Reference Manual*, Intel Corporation, Santa Clara, Calif., 172103-003, 1983.

[In86b] *80386 Programmer's Reference Manual*, Intel Corporation, Santa Clara, Calif., 230985-001, 1986.

[In87] *80386 System Software Writer's Guide*, Intel Corporation, Santa Clara, Calif., 231499-001, 1987.

[In88] *80960MC Programmer's Reference Manual*, Intel Corp., Santa Clara, Calif., 271081-001, 1988.

[In89a] *i486 Microprocessor*, Intel Corporation, Santa Clara, Calif., 240440-001, 1989.

[In89b] *i486 Processor Programmer's Reference Manual*, Intel Corporation, Santa Clara, Calif., 240486-001, 1989.

[Le65] Lett, Alexander S. and William L. Konigsford, "TSS/360: A Time-Shared Operating System", *Proceedings of the Fall Joint Computer Conference*, Vol.30, AFIPS, Thompson Book Co., Washington D.C., 1968, pp. 15–28.

[LeLe83] Leach, Paul J., Paul H. Levine, Bryan P. Douros, James A. Hamilton, David L. Nelson, and Bernard L. Stumpf, "The Architecture of an Integrated Local Network", *IEEE Journal on Selected Areas in Communication*, Vol. SAC-1 #5, IEEE, New York, Nov. 1983, pp. 842–856.

[Li86] Li, Kai, *Shared Virtual Memory on Loosely Coupled Multiprocessors*, Department of Computer Science, Yale University, YALEU/DCS/RR-492, New Haven, Conn., Sep. 1986.

[LiHu90] Li, Kai, and P. Hudak, *Memory Coherence in Shared Virtual Memory Systems*, *ACM Transactions on Computer Systems*, Vol.7 #4, ACM, New York, Nov. 1990, pp. 321–359.

[LiSc89] Li, Kai and Richard Schaefer, *An Operating System Transforming a Hypercube into a Shared-Memory Machine*, Department of Computer Science, Yale University, CS-TR-217-89, Princeton, N.J., April. 1989.

[LiSt88] Li, Kai, Michael Stumm, David Wortman, and SongNian Zhou, *Shared Virtual Memory Accommodating Heterogeneity*, Computer Systems Research Institute, University of Toronto, CSRI-220, Toronto, Dec. 1988.

[Ma76] Madison, A.W. and A.P. Batson, "Characteristics of Program Localities", *Communications of the ACM*, Vol.19 #5, ACM, New York, May 1976, pp. 285–294.

[Ma77] Matick, Richard E., *Computer Storage Systems and Technology*, Wiley-Interscience, New York, 1977.

[Ma82] Madison, Alan Wayne, *Characteristics of Program Localities*, UMI Research Press, Ann Arbor, Mich., 1982.

[MaDo74] Madnick, Stuart E. and John J. Donovan, *Operating Systems*, McGraw-Hill, New York, 1974.

[MaPe75] Manning, Eric and R.W. Peebles, "Segment Transfer Protocols for a Homogeneous Computer Network", *Proceedings ACM SIGCOMM/SIGOPS Interprocess Communication Workshop (Operating Systems Review)*, Vol.9 #3, ACM SIGOPS, New York, 1975, pp. 65–73.

[MaPa89] Mazraani, Tony Y. and Gurudatta M. Parulkar, "Specification of a Multipoint Congram-Oriented High Performance Internet Protocol", *Proceedings of the Ninth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'90)* IEEE Computer Society, Washington D.C., June 1990, abridged from: Washington University Department of Computer Science, technical report WUCS-89-20, St. Louis, Aug. 1989.

[Mo87] *MC68030 Enhanced 32-Bit Microprocessor User's Manual*, Motorola, Inc., Phoenix, MC68030UM/AD, 1987.

[Or72] Organick, Elliot I., *The Multics System: An Examination of Its Structure*, MIT Press, Cambridge, Mass., 1972.

[Or83] Organick, Elliot I., *A Programmer's View of the Intel 432 System*, McGraw-Hill, New York, 1983.

[Pa90] Parulkar, Gurudatta M., "The Next Generation of Internetworking", *Computer Communication Review*, Vol.20 #1, ACM SIGCOMM, New York, Jan. 1990, pp. 18–43, also: Washington University Department of Computer Science, technical report WUCS-89-19, St. Louis, May 1989.

[PaTu90] Parulkar, Gurudatta M. and Jonathan S. Turner, "Towards a Framework for High Speed Communication in a Heterogeneous Networking Environment", *IEEE Network*, Vol.4 #2, IEEE, New York, March 1990, pp. 19–27, also: *Proceedings of the Eighth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'89)*, IEEE Computer Society, Washington, D.C., Vol.II, pp. 655–667, also: Washington University Department of Computer Science, technical report WUCS-88-7, St. Louis, 1988.

[Pl89] Plambeck, K.E., "Concepts of Enterprise Systems Architecture/370", *IBM Systems Journal*, Vol.28 #1, IBM Corporation, Armonk, New York, 1989, pp. 39–61.

[PoWa81] Popek, G., B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudidin, and G. Thiel, "LOCUS: A Network Transparent, High Reliability Distributed System", *Eighth ACM Symposium on Operating Systems Principles (Operating Systems Review)*, Vol.15 #5, ACM SIGOPS, New York, 1981, pp. 169–178.

[PoWa85] Popek, Gerald J. and Bruce J. Walker, *The LOCUS: Distributed System Architecture*, MIT Press, Cambridge, Mass., 1985.

[RaKh88a] Ramachandran, Umakishore and M. Yousef Amin Khalidi, *An Implementation of Distributed Shared Memory*, Georgia Institute of Technology, School of Information and Computer Sciences, technical report GIT-ICS-88/50, Atlanta, Dec. 1988.

[RaKh88b] Ramachandran, Umakishore and M. Yousef Amin Khalidi, *An Evaluation of Memory Management Structures for Object-based Systems*, Georgia Institute of Technology, School of Information and Computer Sciences, technical report GIT-ICS-88/53, Atlanta, Dec. 1988.

[ScGa89] Scalzi, C.A., A.G. Ganek, and R.J. Schmalz, "Enterprise Systems Architecture/370: An Architecture for Multiple Virtual Address Space Access and Authorization", *IBM Systems Journal*, Vol.28 #1, IBM Corporation, Armonk, New York, 1989, pp. 15–38.

[SpOr75] Spier, Michael J. and Elliot Organick, "The Multics Interprocessor Communications Facility", in *Software Systems Principles: A Survey*, Peter Freeman (ed.), Science Research Associates, Chicago, 1975, pp. 133–167.

[StPa89a] Sterbenz, James P.G. and Gurudatta M. Parulkar, *Axon: A High-Speed Communication Architecture for Distributed Applications*, Washington University Department of Computer Science, technical report WUCS-89-36, St. Louis, Sept. 1989, presented at the Fourth IEEE Communications Society Workshop on Computer Communications, Dana Point, California, Oct–Nov 1989.

[StPa89b] Sterbenz, James P.G. and Gurudatta M. Parulkar, *Axon: Application-Oriented Lightweight Transport Protocol Design*, Washington University Department of Computer Science, technical report WUCS-89-14, St. Louis, Sept. 1989.

[St90] Sterbenz, James P.G., *Axon: Host–Network Interface Design*, Washington University Department of Computer Science, technical report WUCS-90-7, St. Louis, March 1990.

[StPa90a] Sterbenz, James P.G. and Gurudatta M. Parulkar, "Axon: Network Virtual Storage Design", *Computer Communication Review*, Vol.20 #2, ACM SIGCOMM, New York, April 1990, pp. 50–65.

[StPa90b] Sterbenz, James P.G. and Gurudatta M. Parulkar, "Axon: A High-Speed Communication Architecture for Distributed Applications", *Proceedings of the Ninth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'90)* IEEE Computer Society, Washington D.C., June 1990.

[StPa90c] Sterbenz, James P.G. and Gurudatta M. Parulkar, "Axon Network Virtual Storage for High Performance Distributed Applications", *10th International Conference on Distributed Computing Systems*, IEEE, Washington D.C., June 1990, pp. 484–492.

[TaFa89] Tam, Ming-Chit and David J. Farber, "CapNet – An Alternative Approach to Ultra-High Speed Network", *International Conference on Communications '90*, IEEE Communications Society, Piscataway, New Jersey, April 1990.

[WaPo83] Walker, Bruce, Gerald Popek, Robert English, Charles Kline, and Greg Thiel, "The LOCUS Distributed Operating System", *Proceedings of the Ninth Symposium on Operating Systems Principles (Operating Systems Review)*, Vol.17 #5, ACM SIGOPS, New York, 1983, pp. 49–70.