

Virtual Active Networks

by
Gong Su

A thesis proposal submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in
Computer Science Department
of the
Columbia University

Spring, 2000

ABSTRACT

This proposal describes the Virtual Active Network (VAN) Architecture. A VAN is a dynamically constructed virtual network that provides application-specific services, such as web caching, multicasting, and transcoding, etc. The goal of a VAN is to enable large scale network applications to control and configure network topology and resources to best support their needs. The key results of this thesis are mechanisms to realize such enabling technology. In particular, the VAN architecture provides: (1) abstractions for applications to specify a VAN; algorithm to map the VAN virtual topology to physical topology; protocols to acquire necessary resources to support the VAN services. (2) algorithm and protocol to resolve deadlock among competitions for shared node and link resources. (3) algorithm and protocol to adapt to physical network failure in order to preserve VAN service semantics.

1 Introduction

The functions and software of current networks are incorporated by vendors in network nodes at design time; the evolution and change of these functions and software is presently a complex and slow process often requiring years of protocol committees work. In sharp contrast, innovative network applications are appearing rapidly, far outpacing the innovations they need from within the network.

Recent DARPA initiative, the active networks research [Tennenhouse, 1996 #11], has recognized the mismatch and seeks to change this by rendering network nodes programmable. An active network is one that supports dynamic deployment and execution of network software in intermediate nodes. This software, such as new protocols, traffic monitoring and filtering software, or resource configuration software, can perform more sophisticated active processing on the application data streams. For example, with an active network one could deploy novel efficient protocols for IP telephony, or deploy firewall functions dynamically to handle topology changes in a mobile network.

The proposed Virtual Active Network (VAN) Architecture is a novel active network architecture that provides features that are necessary for applications to take full advantage of active networks. The goal of a VAN is to enable large scale network applications to control and configure network topology and resources to best support their needs. A VAN is particularly useful for applications that require any of the following services to be provided by the network,

- network coverage of a service
- network location awareness
- specialized QoS support
- high availability and reliability
- security

For example, web caching applications will greatly benefit from services that provide coverage for certain specific locations of the network; and multicasting applications will greatly benefit from specialized QoS support. High availability, reliability, and security are of general interests to any network applications.

The VAN architecture defines abstractions through which applications can specify a virtual active network with desired service type, topology feature, resource constraint, and reliability constraint. The VAN architecture provides algorithms and protocols to dynamically construct such a virtual active network according to application specification. When multiple VANs are being constructed simultaneously and distributedly, potential deadlocks resulting from different VANs' competing for shared node and link resources are resolved by the VAN architecture's conflict resolution algorithm and protocol. At runtime, the VAN architecture monitors underlying physical network condition and adapts accordingly to best keep the VAN semantics when failures, such as a physical link goes down, occur.

The VAN architecture consists of two main functional components that implement a set of algorithms and protocols. The VAN Local Manager (VLM) resides on an active node and manages node-wide resources, such as node processor resource and link resource. The

VAN Domain Server (VDS) typically resides on a node within an administrative domain, aka autonomous system (AS), and coordinates domain-wide resource management through interaction with VLMs within its domain. VDS'es interact with other VDS'es in neighboring domains to coordinate inter-domain resource negotiation.

When an application submits a VAN specification for creation, the VDS serving the application (called the root VDS) contacts other VDS'es to obtain inter-domain topology and resource information. It then invokes the mapping algorithm to compute the virtual topology to physical topology mapping. The mapping, once computed, is disseminated to all the other VDS'es so they can start building their part of the VAN concurrently. Appropriate VNs are created based on the type of VAN within each individual AS by VDS'es interrogating intra-domain VLMs via VISP and inter-domain VLs are negotiated between these VNs by VDS'es coordinating via VESP. When all the other VDS'es finish, they report to the root VDS, which in turn reports to the application. During the construction of a VAN, resource competition between different VANs is resolved by the VAN architecture's priority based preemption algorithm and protocol. At runtime, VLMs monitor the VLs and report to their respective VDS'es when failure occur. The VAN architecture employs "local repair" and "global repair" mechanisms to recover from the failure and to preserve the semantics of the affected VANs as best as it can.

The rest of the proposal is organized as follows. Section 2 illustrates in more detail the functional components of the VAN architecture; Section 3 describes the abstractions for specifying a VAN, the distributed algorithm and protocol for mapping virtual topology to physical topology, and the protocol for implementing the mapping; Section 4 presents the algorithm and protocol for preventing resource deadlock when VANs are being constructed simultaneously and distributedly; Section 5 introduces the monitoring and adaptation mechanisms to best keep the semantics of VANs when link failures occur; Section 6 presents a prototypical implementation of VAN; Section 7 compares the VAN architecture to related work in this area; and Section 8 lays out the schedule for the completion of the thesis.

2 VAN Architecture Functional Components

Before we present the novel results achieved by the VAN architecture, it is necessary to briefly describe the main functional components of the VAN architecture and their relation to each other. It is the interplay among these components that fosters these novel results. A high-level diagram depicting these components is shown below.

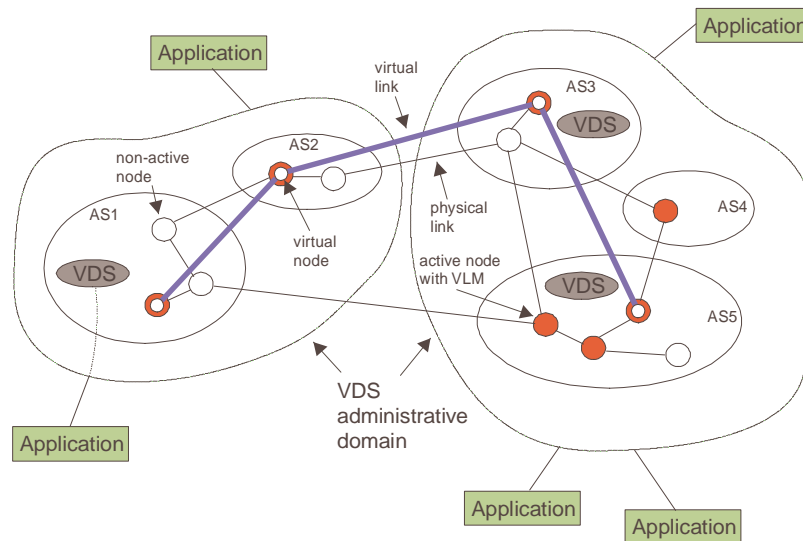


Figure 2-1: VAN Architecture Functional Components

2.1 Virtual Node (VN) and Virtual Link (VL)

VNs are active packet processing applications running on active nodes. VNs provide application-specific packet processing functionality inside the network. Examples of such active applications are web caching, multicast routing, and video transcoding, etc. VNs are processes instantiated by VAN Local Manager (Section 2.2) on-demand at runtime according to application requests.

In a distributed application such as web caching and multicasting, many VNs will cooperate with each other and collectively achieve their intended goals. VLs are datalink layer logical communication channels that provide the means for VNs to communicate with each other. The need for a datalink layer channel is due to the fact that in active networks the assumption of a common network layer, such as IP in the traditional network, is no long valid. Active packet processing applications may perform functions anywhere from network layer to application layer. Essentially, data link layer becomes the lowest common layer. As an example, web caching applications usually employ their own routing scheme to route the URL requests amongst the mesh of caching applications. In other words, peer web caching applications see each other as a direct datalink layer connected peer, rather than a network layer connected peer.

VLs can be built on top of physical datalink layer connecting two VNs residing on the same

datalink layer network, such as an Ethernet LAN. VLs can also be tunneled through a network layer, such as IP, connecting two VNs spanning across a wide area network.

2.2 VAN Local Manager (VLM)

VLMs are the software running on active nodes deployed inside the network. VLMs extend the Operating System for managing node resources on the active nodes, such as coordinated scheduling of processor cycles and link bandwidth. VLMs export the following services to VDS'es (Section 2.3) and VNs running on the local active node:

- create/delete VNs with guaranteed processor resource
- create/delete VLs with guaranteed link resource
- modify VN and VL resources

VLMs also export the following additional services to the VNs:

- create/delete virtual ports (VPs)
- bind/unbind VPs to/from VLs
- send/receive messages to/from VPs

The figure below depicts the functional components of a VLM. Note that although the figure separates userspace and kernel space, it by no means indicates that this is the only way to implement a VLM. In fact, an implementation may choose to put all the components in the kernel space to achieve higher performance.

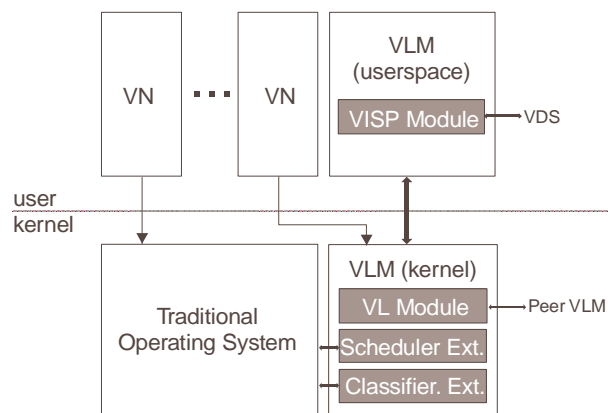


Figure 2-2: VLM Functional Components

- **VISP Module** implements the VAN Interior Setup Protocol, which is the protocol used by VDS'es for interacting with VLMs for resource acquisition and releasing
- **VL Module** implements the VL signaling and tunneling setup protocol between peer VLMs
- **Scheduler Extension** cooperates with OS scheduler to guarantee and regulate VN processor resource and VL link resource
- **Classifier Extension** extends OS packet classifier to support VL multiplexing and demultiplexing

2.3 VAN Domain Server (VDS)

VDS'es perform domain-wide resource management functionality. A VDS may manage resources of multiple AS'es as shown in Figure 2-1. Multiple VDS'es may also manage one AS. So the association of a VDS with an AS is logical. VDS'es serve two important functions of the VAN architecture. First, VDS'es interface with applications to export the services that enable applications control and configure VANs. VDS'es implement the algorithm and protocol for mapping VAN virtual topology specified by the application to underlying physical topology. The following services are exported to applications:

- create/delete VANs
- join/leave VANs

Second, VDS'es coordinate intra- and inter-domain resource acquisition for constructing VANs, resolving conflict as appropriate. VDS'es interrogate VLMs within its administrative domain via VISP to coordinate intra-domain resource negotiation. VDS'es cooperate with other VDS'es in neighboring domains via VESP (VAN Exterior Setup Protocol) to coordinate inter-domain resource negotiation.

The following figure illustrates the functional components of VDS'es. Again, the picture does not intend to imply any restriction on the actual implementation of VDS'es.

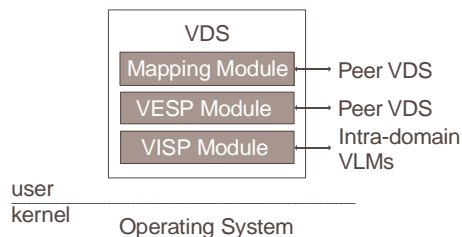


Figure 2-3: VDS Functional Components

- **Mapping Module** implements the distributed algorithm and protocol for mapping a virtual topology onto a physical topology
- **VESP Module** implements the VAN Exterior Setup Protocol for inter-domain resource negotiation
- **VISP Module** implements the VAN Interior Setup Protocol for intra-domain resource negotiation

2.4 General Assumptions

The VAN architecture makes certain general assumptions about the system environment within which it operates. These assumptions are made because they are issues orthogonal to the architecture itself.

1. There is some way of representing resource types and usage. It is assumed that hardware and software resources, such as CPU cycle, memory, bandwidth, and code for active application, etc., as well as the usage of these resources, can be represented in

certain canonical way for general purpose computation. Such general purpose computation may include query resource type and usage, reserve and relinquish portions of the resources, etc.

2. There is some way of maintaining domain-level topology and resource availability information. It is assumed that there is a database of current resource usage for each VDS administrative domain maintained by an entity. This entity can be a VDS itself or a separate entity. This entity employs necessary mechanisms to keep the database up-to-date (e.g., periodically query VLMs) and to arbitrate the access to the database.
3. There is some external mechanism, such as a directory service, that when given an AS will return the VDS that is responsible for the AS.

3 Virtual Topology Specification and Mapping

In order to facilitate provision of application-specific network services, there must be mechanisms to allow applications to specify their needs, to translate their needs into underlying physical resource requirement, and to acquire these resources to support their needs. In this section, we introduce the following mechanisms provided by the VAN architecture to address these issues: (1) abstractions for specifying a VAN; (2) virtual topology to physical topology mapping algorithm; and (3) intra-domain and inter-domain resource acquisition protocols.

3.1 VAN Specification

The VAN architecture defines a set of abstractions in terms of which applications can specify a VAN of interconnected AS'es with desired service type, topology features, resource constraint, and reliability constraint. The abstractions are formulated based on graph theory [Harary, 1969 #116] and are listed below,

1. Service type. Indicates the application-specific type of processing inside the network. For example, an application may specify a VAN for web caching, or a VAN for multicasting.
2. Topology features. Specifies desired virtual topology features of interconnected AS'es.
 - Coverage: AS[i], $i=1, \dots, n$. It is assumed that all AS'es must be connected.
 - Connectivity: there are two ways to specify connectivity features. One is to use the following abstractions,
 - degree(AS[i]), $i=1, \dots, n$ (number of VLs connected to each AS)
 - cyclic or acyclic
 - diameter (maximum distance between any two AS'es)
 Examples of common topologies that can be easily specified with these abstractions:
 - i. Chain: acyclic and degree(AS[i]) \leq 2 for all i
 - ii. Ring: degree(AS[i])=2 for all i
 - iii. Star: degree(AS[i])=1 for all i except one of them
 - iv. Tree: acyclic
 - v. Clique: degree(AS[i])=n-1 for all i
 With more "irregular" topology, this abstraction becomes cumbersome. Another way of specifying the desired topology is with the "anonymous" graphs such as those in Figure 3-1. Note that applications do *not* specify explicitly which AS will be which anonymous node in the graph, i.e., A, B, C, and D can be any permutation of the AS'es. In addition, applications have the flexibility to partially specify the topology such as the graph to the right in the figure, which the application has specifically chosen to put AS2 at the "C" position.

3. Resource constraint. For all AS'es, the processor cycles needed to support application-specific processing inside the network; for all VLs, the link bandwidth desired.

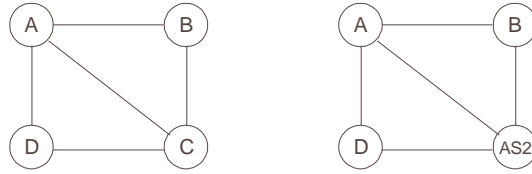


Figure 3-1: Anonymous Graph for Virtual Topology

4. Reliability constraint. Reliability is expressed in terms of the number of different VLs a physical link carries. Intuitively, the more VLs a physical link carries, the less reliable the VAN is since the failure of the physical link will bring down all the VLs. So applications may specify an upper limit on the number of VLs in a VAN that traverse any particular physical link.

Note that a VAN is specified at the AS level, i.e., a VAN is a virtual active network of interconnected AS'es. The reason for this is that given the sheer size and complexity of today's network no application can ever have enough knowledge about the network at the level of individual devices for it to specify a virtual topology at this level. The abstractions are thus designed to allow applications to specify a VAN at the AS level and also to concentrate on the *features* of the VAN rather than the details of how exactly each AS should be connected. For example, applications only need to specify that they need a star comprising of AS1, AS2, AS3, and AS4 but not which AS should be the center. The philosophy here is that even at AS level applications may not have enough knowledge about the network to specify the virtual topology down to the detail that how exactly each AS should be connected. It is the VAN architecture's responsibility to best map this virtual (star) topology to the physical topology according to the constraints specified by the application. Also, applications are more likely to be interested in the features of the topology rather than its details even if they have enough knowledge about the network to specify the topology in detail (in which case they can still do so with the "anonymous" graph).

3.2 Virtual Topology to Physical Topology Mapping

Given a VAN specification submitted by an application, the VAN architecture mapping algorithm is responsible for translating the specification into the physical resource requirement that provides the service type and topology feature while satisfying the resource and reliability constraints demanded by the applications. We first mathematically formulate the mapping problem and then discuss possible heuristic algorithms to solve the problem.

3.2.1 Problem Definition

1. A physical network is defined as an undirected graph $P=(V_P, E_P)$ where V_P is the set of nodes (AS'es in our case) and E_P is the set of edges. For each $v_p \in V_P$ let $PC_p(v_p)$ be the available processing capacity of v_p ; and for each $(v_p, w_p) \in E_P$ let $LC_p(v_p, w_p)$ be the available link capacity of (v_p, w_p) . Let V_S be any subset of V_P i.e., $V_S \subseteq V_P$

2. A virtual network is defined as an undirected graph $Q=(V_Q, E_Q)$ such that $|V_Q|=|V_S|$. For each $v_q \in V_Q$, let $PC_q(v_q)$ be the desired processing capacity of v_q ; and for each $(v_q, w_q) \in E_Q$, let $LC_q(v_q, w_q)$ be the desired link capacity of (v_q, w_q) .
3. A mapping of Q to P is a function $f:V_Q \rightarrow V_S$ such that: (1) for all $v_q \in V_Q$ and $w_q \in V_Q$, $f(v_q)=f(w_q)$ if and only if $v_q=w_q$; (2) for all $(v_q, w_q) \in E_Q$, there is a path $p(f(v_q), f(w_q)) \subset E_P$ in P between $f(v_q)$ and $f(w_q)$. Let $E=\cup p(f(v_q), f(w_q))$ for all $(f(v_q), f(w_q))$ pair. Let $r(e)$ be the number of times e appears in different $p(f(v_q), f(w_q))$ for all $e \in E$.

With the above definition, the statement of the optimization problem we are trying to solve can be presented as follows:

Given a physical network P and a virtual network Q , find a mapping function f such that the following constraints hold,

1. For all $v_q \in V_Q$, $PC_q(v_q) \leq PC_p(f(v_q))$. This is the processor capacity constraint.
2. For all $(v_q, w_q) \in E_Q$, $LC_q(v_q, w_q) \leq \min\{LC_p(e)\}$ for all $e \in p(f(v_q), f(w_q))$. This is the link bandwidth constraint.
3. $\max\{r(e)\} \leq r_Q$ for all $e \in E=\cup p(f(v_q), f(w_q))$, where r_Q is the reliability constraint number. This is the reliability constraint.

3.2.2 Mapping Algorithm

We describe a simple heuristic algorithm here for mapping a virtual topology to a physical topology while satisfying the resource and reliability constraints. Note that it may not be possible to generate such a mapping with all the constraints, in which case the algorithm will present a “the best you can get” mapping.

The algorithm maintains two graph data structures, one for the virtual topology, the other for the physical topology. It first maps the virtual nodes to the physical nodes (remember that this is a one to one mapping). It then maps the virtual links to physical paths. The following is the main steps taken by the algorithm.

- (1) Sort all virtual nodes and physical nodes by their degree in descending order. Nodes with the same degree are ordered arbitrarily.
- (2) Map virtual nodes to physical nodes one to one according to the sorted order, i.e., the highest degree virtual node is mapped to the highest degree physical node, etc.
- (3) Scan the physical topology and mark the links that don't have enough available link capacity for the virtual links as infeasible. No virtual link will be mapped to a physical path traversing an infeasible physical link. If the physical topology consisting of only feasible links is partitioned, stop and declare that no mapping can be produced.

- (4) Associated with each feasible physical link is a counter, initially 0, which counts the number of times it has been mapped onto, i.e., the number of virtual links it is carrying. Pick a virtual link, map it to a physical path between the two physical nodes (where the two virtual nodes of the virtual link are mapped onto) such that the highest counter of the feasible physical links traversed by the path is minimized.
- (5) Increment the counter of all the physical links in (4) and subtract the capacity of them by the capacity of the virtual link. If any of these physical link is left with capacity lower than the capacity of a virtual link, mark it as infeasible.
- (6) Repeat (4) until all virtual links are mapped or stop when a virtual link can not be mapped.

We intend to study the behavior of this heuristic with optimization techniques, such as different ways of mapping virtual nodes to physical nodes in (1) and (2), precomputing all physical edge-disjoint paths between any two physical nodes, changing the order of picking the virtual link in (4), etc. We also intend to explore the possibility of other heuristics.

3.3 Resource Acquisition Protocols

Once the virtual topology to physical topology mapping is computed, the VDS'es will use the VAN Interior Setup Protocol to acquire node and link resources from interacting with the VLMs in their respective domain. For VLs crossing VDS administrative domains, the VDS'es at both ends of the VLs will coordinate through VAN Exterior Setup Protocol to negotiate the link resources.

3.3.1 VAN Interior Setup Protocol (VISP)

VISP is used by VDS'es and VLMs in the same domain to negotiate intra-domain resource acquisition. The protocol messages for setting up VNs and intra-domain VLs are shown in Figure 3-2. More protocol messages for resolving conflict are shown Figure 4-2 in Section 4.1.3.

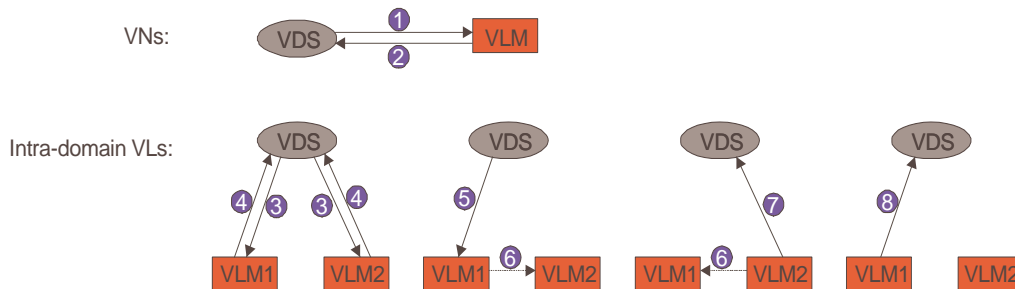


Figure 3-2: VAN Interior Setup Protocol

- (1) VISP_VN_INIT/VISP_VN_FREE: request for creation/deletion of a VN with certain processor cycles.
- (2) VISP_VN_INITACK/VISP_VN_FREEACK: reply to VN creation request.
- (3) VISP_VL_RESV: request for reservation for a VL.
- (4) VISP_VL_RESVACK: reply to reservation request.
- (5) VISP_VL_INIT/VISP_VL_FREE: request initiation of setting up/tearing down a VL. Messages (3) and (4) are only for VISP_VL_INIT.
- (6) Signaling and tunneling setup/tear down for the VL.
- (7) VISP_VL_UP/VISP_VL_DN: notify VDS about a VL is up/down.
- (8) VISP_VL_INITACK/VISP_VL_FREEACK: reply to VL setup/tear down request.

3.3.2 VAN Exterior Setup Protocol (VESP)

VESP is used by VDS'es in different administrative domains to query and negotiate inter-domain topology and resources. More protocol messages for resolving conflict are shown Figure 4-2 in Section 4.1.3.

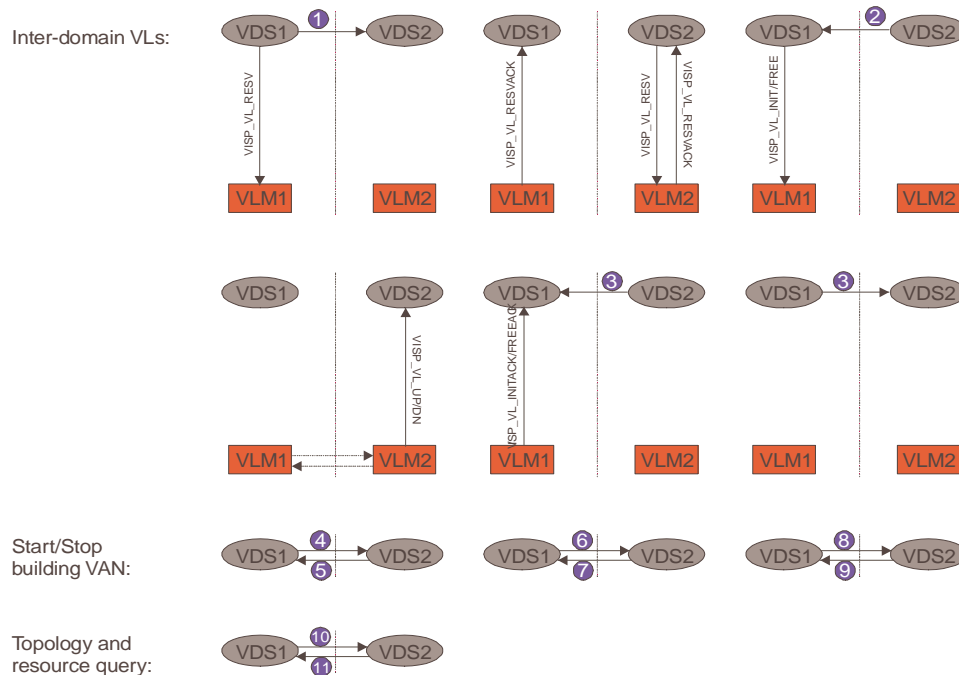


Figure 3-3: VAN Exterior Setup Protocol

- (1) VESP_VL_RESV: request reservation for an inter-domain VL. The problem with creating an inter-domain VL is that VDS1 does not know which physical node (managed by VLM2) the other VN is mapped to by VDS2 in the other domain. This message will serve to request for both a reservation and a location of the VLM2.
- (2) VESP_VL_RESVACK: reply to reservation request. This message will contain the location of VLM2 so VLM1 will know who to contact.
- (3) VESP_VL_UP/VESP_VL_DN: notify peer VDS an inter-domain VL is up/down.
- (4) VESP_VAN_INIT: request initiation of building a VAN.
- (5) VESP_VAN_INITACK: reply to VESP_VAN_INIT.
- (6) VESP_VAN_FREE: request initiation of tearing down a VAN.
- (7) VESP_VAN_FREEACK: reply to VESP_VAN_FREEACK.
- (8) VESP_VAN_UPD: incremental change to a VAN (see Section 5.2.2)
- (9) VESP_VAN_UPDACK: reply to VESP_VAN_UPD.
- (10) VESP_RES_QUERY: query inter-domain topology and resource information.
- (11) VESP_RES_QUERYACK: reply to VESP_RES_QUERY.

4 Conflict Resolution for Resource Acquisition

Building a VAN is a distributed resource acquisition process. Competition for shared resources between VANs being built simultaneously will occur. This is somewhat similar to distributed database transactions [Gray, 1992 #117] where node and link resources are analogous to databases, VLMs are to database managers, and VDS'es are to transaction managers. The key differences, however, between the two are: (1) constructing a VAN usually involves large number of sites, which makes traditional distributed deadlock detection algorithms infeasible since most of these algorithms have message complexity of $O(n^2)$ and detect delay of $O(n)$ [Singhal, 1994 #118]; (2) the "transactions" of building a VAN, i.e., acquiring node and link resources, are fairly simple and uniform operations comparing to typical database transactions, which makes traditionally infeasible deadlock prevention with preemption viable due to the reduced rollback cost of the preempted "victim".

In this section, we will first discuss a possible approach for preventing deadlock which is based on the global total ordering of resources [Havender, 1968 #119]. We will discuss its drawbacks and introduce another deadlock prevention algorithm that is based on preemption [Rosenkrantz, June 1978 #120] using priority dynamically assigned to a VAN. We will present the algorithm and the protocol, explain the rationale behind it, and compare it to some of the traditional distributed deadlock detection algorithms and protocols.

4.1 VAN Deadlock Prevention Algorithm and Protocol

4.1.1 Deadlock Prevention by Global Resource Ordering

One way to prevent deadlock is to impose a global total ordering on the AS'es; and within each AS a total ordering of the node and link resources. VANs are required to be built sequentially following the order of AS numbers. And resources are allocated on a First Come First Serve basis. A VAN that can not acquire a needed resource may search for other (higher numbered) resources or wait for a finite amount of time before giving up and releasing all the resources it has acquired so far to let other VANs have a chance to proceed.

The advantage of this approach is its simplicity. However, it does have a couple of drawbacks. First, it limits the concurrency of building a VAN as a VAN must be built sequentially from AS to another. Second, starvation situation can occur since different VANs start from different AS'es. If enough VANs keep starting from a higher numbered AS, a VAN starts from a lower numbered AS may never get a chance to acquire the resources it needs from the higher numbered AS. This problem can potentially be fixed by associating a timestamp with each VAN and *not* reset the timestamp when a VAN is restarted. This essentially assigns each VAN with a static priority (the timestamp) for arbitration when conflict occurs.

Due to these inadequacies of global ordering, we explore another approach which is based on the preemption using priority. However, instead of using a fixed priority for each VAN being built, our algorithm dynamically computes what we call the "Progress Index" (PI) for each VAN and use it as the priority for preemption.

4.1.2 Progress Index Algorithm

The PI is a number that is computed to indicate how far along a VAN is into its building process. It is used to dynamically assign priority to VANs in order to resolve conflict through preemption. The idea is that a VAN that has finished most parts of it (also taking into consideration of the total size of the VAN) should be given higher priority.

As a VAN is built distributedly by many VDS'es, each VDS will compute a "local total weight" and a "local current weight" for its part of the VAN as follows:

1. A "weight" is associated with each VN and VL. This weight assignment should be consistent across all VDS'es. Otherwise, VANs built by VDS'es that assign "better" weight value to VNs and VLs will have unfair advantage over VANs built by other VDS'es.
2. The "local total weight" of the part of the VAN that lies within the VDS administrative domain is simply the sum of the weight of all the VNs and VLs to be built by the VDS.
3. For each VN or VL successfully built, its weight is added to the "local current weight" of the VAN.

The "global PI" (GPI) of a VAN is computed based on the "global total weight" of the VAN, which is the sum of the "local total weight" of all VDS'es, and the "global current weight" of the VAN, which is the sum of the "local current weight" of all VDS'es. The exact algorithm for computing the GPI is to be determined.

4.1.3 Conflict Resolution Protocol

The rule used by VLMs for arbitrating conflicts is fairly simply. When the resource for a VN or VL is requested by a VDS, the protocol message carries the GPI of the VAN to which the VN or VL belongs. When conflict occurs, whichever VN or VL that has the higher GPI wins. When there is a tie, pick one randomly. The preempted VAN has the choice of either waiting (optionally with time-out) for the resource to become available or attempting to find alternative resource.

We now show that it is not enough to use just a "local PI" (LPI, which is computed from the "local total weight" and "local current weight") within each VDS domain to resolve the conflict. It would work if a VAN were only spanning one VDS administrative domain. But when a VAN is built simultaneously in multiple domains and conflicts between different VANs occur in different domains, we have a problem - as illustrated in the figure below.

Shown in the figure are two VDS administrative domains in which two instances of VDS'es are building two different VANs concurrently. VDS1 in domain A and VDS3 in domain B are building *VAN1*; VDS2 in domain A and VDS4 in domain B are building *VAN2*. Assuming that the VLMs will use LPI to resolve the conflict within their respective domain. In domain A, VDS1 computes its LPI for *VAN1* as 3 and VDS2 computes its LPI for *VAN2* as 6. And in domain B the numbers are 7 for *VAN1* and 2 for *VAN2* respectively. It is clear that *VAN1* will win the battle in domain B but will lose in domain A, and vice versa for *VAN2*. This results in a deadlock or livelock (depending on the behavior of the loser) across

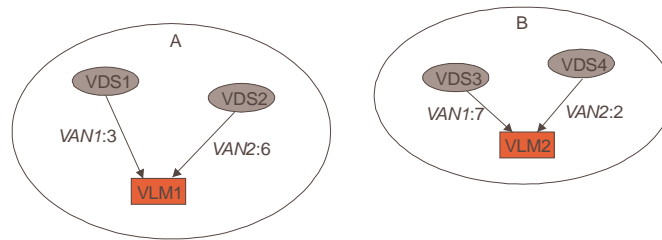


Figure 4-1: Deadlock With Global Conflict

domains between A and B. In essence, the problem is that there is no total ordering of LPI. We present a solution to remedy this situation.

The basic idea of the solution is to synchronize the LPIs of all the VDS'es building the same VAN in different domains at the time of conflict. That is, upon detecting a conflict by a VLM, instead of using the LPI for the arbitration, the VLM will inform the conflicting VDS'es to start a GPI synchronization protocol. The figure below describes the protocol interaction. Only the conflict resolution in domain A is shown. Domain B will carry out the same procedure.

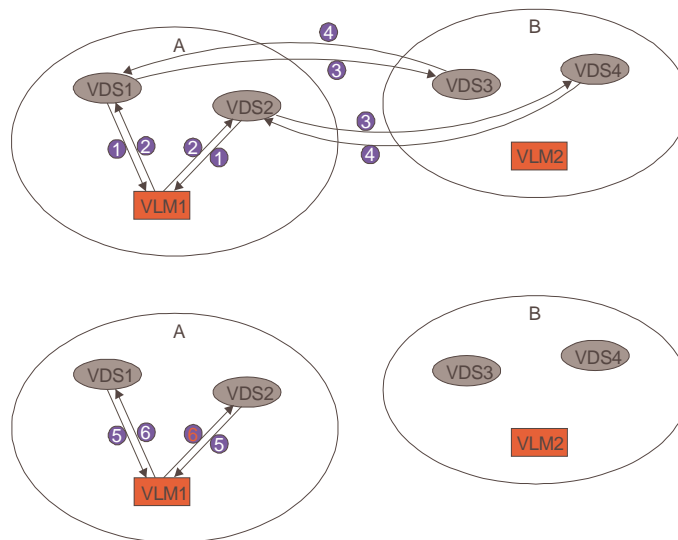


Figure 4-2: Conflict Resolution Protocol

- (1) VDS'es send VISP_VN_INIT or VISP_VL_RESV to acquire VN or VL resources.
- (2) VLM detects conflict and notifies VDS'es with VISP_RES_ARBT that an arbitration decision needs to be made.
- (3) VDS'es broadcast VESP_PI_SYNC to all other VDS'es in their respective VAN requesting synchronization of GPI.

- (4) VDS'es reply to sender of VESP_PI_SYNC with their "local total weight" and "local current weight" included in VESP_PI_SYNCACK.
- (5) VDS'es compute the GPI after getting all the replies and send VISP_RES_ARBTACK with the GPI to VLM.
- (6) Based on the now synchronized GPI, VLM decides the winner and the loser. The winner will receive VISP_VN_INITACK/VISP_VL_RESVACK with positive answer and the loser will receive the same message with negative answer.

It is clear that since the arbitration decision is made based on the GPI, which will be the same for domain A and B after carrying out the conflict resolution protocol (10 for VAN1 and 8 for VAN2), domain A and B will make the same decision, i.e., either VAN1 or VAN2 will be the winner and the other will be the loser. Situation like the one we described in Figure 4-1 will not occur.

4.1.4 Potential Drawbacks of Our Algorithm and Protocol

One potential problem with our protocol is the implosion of VESP_PI_SYNCACK messages towards the sender of VESP_PI_SYNC. A possible solution to mitigate the problem is to logically organize all the VDS'es for a VAN into a spanning tree and conduct our protocol over the spanning tree. This way, a sender broadcasts VESP_PI_SYNC to all its neighbors in the spanning tree and act as the root of the spanning tree for the duration of the conflict resolution. The receivers, i.e., the descendants, will in turn broadcast VESP_PI_SYNC to all their descendants. When sending back VESP_PI_SYNCACK, each VDS will aggregate the "partial total weight" and "partial current weight" from all its descendants before reporting to its predecessor. When the VESP_PI_SYNCACK comes back from all its descendants, the root can compute the "total weight" and "current weight" of the VAN by simply aggregating the result from its descendants. This solves the implosion problem but also incurs a delay in the conflict resolution process, which is $O(d)$ where d is the diameter of the spanning tree. There is also some overhead in maintaining the spanning tree.

Another problem our protocol (and any preemption scheme) must deal with is the starvation problem. That is, no VAN should be preempted indefinitely and thus could never finish. This problem depends on the algorithm of computing the GPI, i.e., the priority, of the VANs. We will investigate good algorithms for computing the GPI to prevent such situation from occurring.

4.2 Comparison with Traditional Algorithms and Protocols

Traditional distributed deadlock detection algorithms can be divided into four classes: path-pushing, edge-chasing, diffusion computation, and global state detection. We use one example from each class.

1. **Path-pushing.** The wait-for dependency information of the global WFG is disseminated in the form of paths, i.e., a sequence of wait-for dependency edges. A classic example of path-pushing algorithm is Obermarck's algorithm [Obermarck, June 1982 #121]. The algorithm sends $n(n-1)/2$ messages to detect a deadlock involving n sites. Size of a message is $O(n)$. The delay in detecting the deadlock is $O(n)$.
2. **Edge-chasing.** Special messages of fixed size called probes are circulated along the edges of the WFG to detect a cycle. When a blocked process receives a probe, it propagates the probe along its outgoing edges in the WFG. A process declares a deadlock when it receives a probe initiated by itself. An example of edge-chasing algorithm is Chandy-Misra-Hass's algorithm [Chandy, May 1983 #122]. The algorithm exchanges at most $m(n-1)/2$ messages to detect a deadlock that involves m processes and spans over n sites. The size of the messages is fixed and very small. The delay in detecting the deadlock is $O(n)$.
3. **Diffusion computation.** A process determines if it is deadlocked by initiating a diffusion computation. The messages used in diffusion computation take the form of a query(i, j, k) and a reply(i, j, k), denoting that they belong to a diffusion computation initiated by a process P_i and are being sent from process P_j to process P_k . The algorithm defines the action of process P_k upon receiving query(i, j, k). An initiator detects a deadlock when it receives reply messages to all the query messages it had sent out.
4. **Global state detection.** A consistent snapshot of a distributed system is obtained without freezing the underlying computation. The consistent snapshot may not represent the system state at any moment in time, but if a stable property (e.g., deadlock) holds in the system before the snapshot collection is initiated, this property will still hold in the snapshot. An example of global state detection algorithm is Kshemkalyani-Singhal algorithm [Kshemkalyani, 1990 #123]. The algorithm has a message complexity of $4e-2n+2l$ and a delay time complexity of $2d$ hops, where e is the number of edges, n the number of nodes, l the number of leaf nodes, and d the diameter of the WFG.

Our protocol has a message complexity of $O(n)$ and delay time complexity of $O(1)$ with fixed-size messages, where n is the number of sites.

5 Failure Adaptation

A virtual link may traverse multiple physical links; also many virtual links may share the same physical link. It is thus imperative that when a physical link fails, the VAN architecture must adapt to preserve the original semantics (i.e., service type, topology feature, resource constraint, and reliability constraint) of the VANs affected; and it must do so as effectively as possible. This section presents the VAN architecture's monitoring and adaptation mechanisms to deal with physical link failures.

5.1 VL Monitoring

The two VLMs at each end of a VL uses a `VISP_VL_KEEPAKIVE` message to refresh the resource reserved for the VL along its path and to detect failure of the VL (e.g., one of the physical links in the path of the VL goes down). Upon detecting a VL failure, VLMs notify their respective VDS'es with the `VISP_VL_DN` message. If the VL in an inter-domain one, additional messages (`VESP_VL_DN`) are exchanged between the involved VDS'es. The figure below describes the interactions.

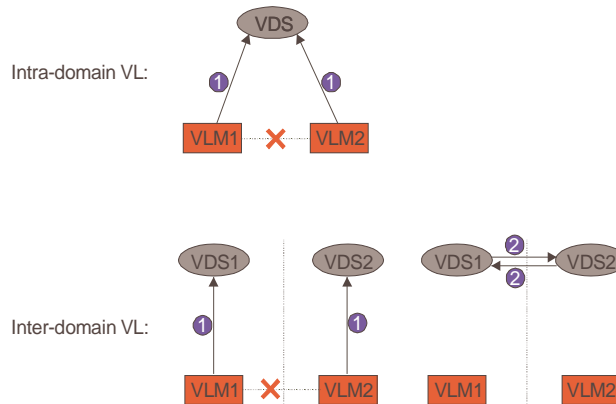


Figure 5-1: VL Monitoring

- (1) `VISP_VL_DN`: used by VLMs to notify VDS'es about VL failure.
- (2) `VESP_VL_DN`: user by VDS'es to notify each other about an inter-domain VL failure.

5.2 Failure Adaptation with Local Repair and Global Repair

5.2.1 Local Repair

In order to recover lost VLs due to a failed physical link as quickly as possible, the first logical thing to do would be to locally find an alternative path between the two disconnected VNs. For an intra-domain VL, the VDS for the domain in question recomputes an alter-

native path between the two VNs based on its local topology and resource information of the domain. For an inter-domain VL, the two VDS'es involved use the topology and resource information between them to recompute a new path for the lost VL.

The advantage of local repair is that it preserves the original VAN topology and it recovers relatively quickly since lost VLs are repaired locally and only the lost VLs are affected. However, the problem with local repair is that since it uses only local topology and resource information, the recomputed new path may traverse physical links that are already serving other VLs of the VAN. This may result in the violation of the reliability constraint of the VAN specified by the application. In addition, when an alternative path can *not* be found between the disconnected VNs, there is nothing local repair can do. But as we see from the figure below, it is certainly possible to reconstruct the VAN topology even when local repair fails. In the figure a tree is disconnected due to the failure of the VL between AS2 and

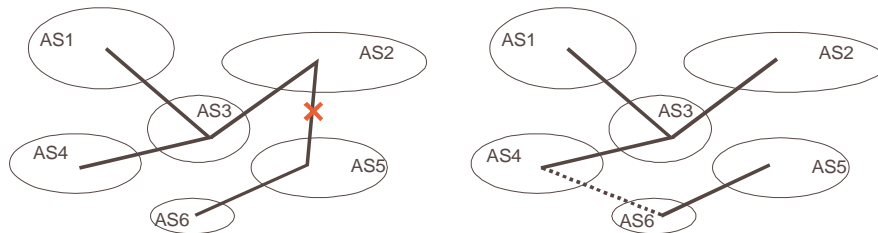


Figure 5-2: Local Repair Inadequacy

AS5. Assuming no other physical path exists between AS2 and AS5, local repair would leave the tree topology in limbo. However, there may well be another physical path between AS4 and AS6 so that a VL can be established between them to restore the tree topology. Note that the new tree is *not* the same tree as the old one. But recall Section 3.1 that topology is specified in terms of coverage and connectivity features rather than details of which AS should be connected to which other AS('es).

5.2.2 Global Repair

When local repair violates the reliability constraint of the VAN or when no alternative path can be found between the two VNs of the failed VL, global knowledge is needed to best preserve the reliability and topology constraint of the VAN. This is done by reporting the failure to the root VDS of the VAN which has the necessary global information to recompute the mapping. The sequence of protocol messages exchanged among the local VDS'es (where VL failure happens) and the root VDS are shown in the figure below.

- (1) VESP_VL_NOPATH: when an inter-domain VL goes down, one of the VDS'es (e.g., the one with a lower IP address) will be responsible for recomputing the new path. If a new path can not be found, this message is sent to the other VDS. Note that the failed physical link may actually also cut off the communication between the two VDS'es. In this case, both VDS'es will time-out (VDS2 does not see

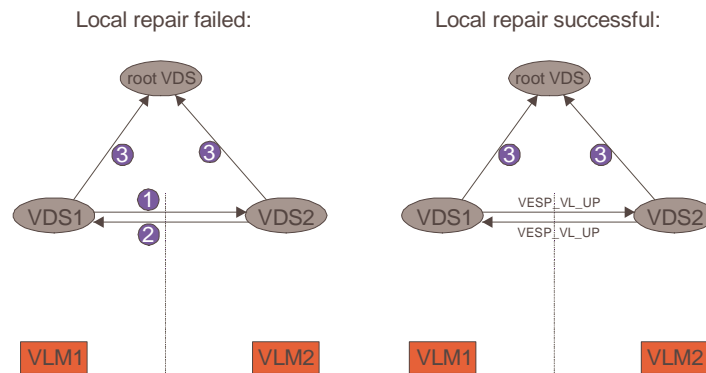


Figure 5-3: Global Repair Protocol Message Exchange

VESP_VL_NOPATH and VDS1 does not see VESP_VL_NOPATHACK) after a while and both proceed to declare the VL fully down and can not be reconstructed locally.

- (2) VESP_VL_NOPATHACK: reply to VESP_VL_NOPATH.
- (3) VESP_MAP_UPD: whether or not a local repair is successful, VESP_MAP_UPD is sent to the root VDS to signify the condition of a VL mapping change. If the local repair failed or if the succeeded local repair violates the reliability constraint of the VAN, the root VDS recomputes a new VL mapping using the global topology and resource information. The new VL mapping is sent along with the VESP_VAN_UPD to appropriate VDS'es while the VDS'es of the locally repaired VL will be notified to tear the VL down.

Global repair uses global topology and resource information to recompute new paths for failed VLs. It can therefore reconstruct the topology features of the VAN whilst satisfying the resource and reliability constraints when local repair fails. However, it does so at the cost of storage and computation overhead; the communication between the root VDS and the local VDS'es also incurs delay. The computational overhead may be tackled by designing algorithms that can do incremental mapping of virtual topology to physical topology while satisfying the constraints, i.e., the algorithm does not have to recompute the whole mapping again. We will explore the possibility of extending our mapping algorithm described in Section 3.2.2 with such capability.

5.2.3 Combine Both Approaches

Given the pros and cons of local repair and global repair respectively, it is clear that neither of them alone can resolve the failure problem satisfactorily. Therefore, the VAN architecture combines both approaches and tries to make the best of them. The strategy is that when a VL fails due to a physical link failure, a local repair is attempted first to *temporarily* restore the VL as quickly as possible. Meanwhile, the root VDS is notified to validate the temporary VL with the reliability constraint and to recompute a new permanent VL if nec-

essary. In essence, local repair and global repair work in concert to compensate for the inadequacy of each other and together they make the failure adaptation as smooth as possible.

6 Prototypical Implementation

To evaluate the feasibility of the ideas in VAN, we have implemented and demonstrated a prototype VAN to illustrate some of the basic constructs of VAN, which includes,

- the VN, VL, and VP objects;
- APIs for dynamically creating, deleting, and configuring these objects;
- APIs for dynamically loadable VN Engines;
- APIs for dynamically loadable VL Engines;
- protocol and APIs for managing (creating, deleting, configuring, and monitoring) a VAN;
- experimental VN Engines that provide traffic redirection and IP routing (RIP capable);
- experimental VL Engines that provide UDP tunnelled VL and QoS guaranteed (via RSVP) VL;
- front-end command line tool with scripting capability for managing a VAN;
- interface with SNMP agent to allow monitoring of VAN with a GUI (Smarts InCharge);
- simple plaintext password based authentication and host based authorization.

The prototype is built on Linux platform acting as both end hosts and intermediate routers. Currently, the prototype can construct a VAN on top of the IP network with an arbitrary topology. Figure 6-1 shows an example. The topology of the VAN can be changed dynamically.

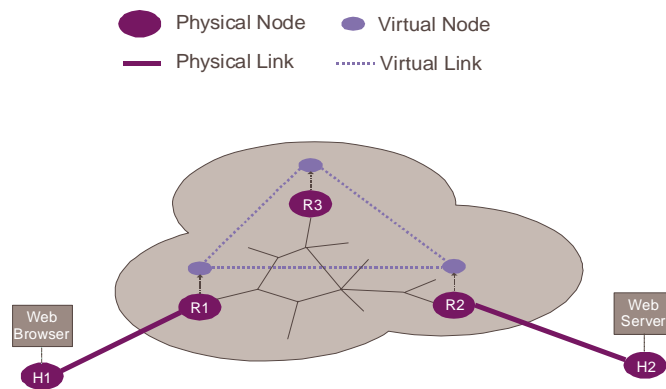


Figure 6-1: Demo Virtual Topology

The front-end command line tool allows a management station to create a VAN remotely based on a script that describes the desired configuration of the VAN. The script specifies the VNs and VLs to be created and how they are combined to give the desired VAN topology. It also specifies what VN Engine and VL Engine to load into each VN and VL object. When executed, the script is translated by the command line tool into appropriate VAN APIs to construct the whole VAN. Our current prototype does not yet instrument the virtual to physical topology mapping capability but the system is design in a modular way that a mapping algorithm can be “dropped in” when it is available.

At the boundary of the VAN and the IP network (such as R1 and R2 in Figure 6-1), a VN

with a special VN Engine is used to trap IP network traffic and redirect it into the VAN, and vice versa. In the VAN, the traffic is processed by the customized VN Engine in each VN. These VN Engines can be dynamically programmed to change their behavior by the management station. We have implemented VN Engines that can perform IP routing. When deployed into these VNs, we effectively construct a virtual IP network with arbitrary topology on top of the real IP network. The virtual links can have simple QoS through RSVP. The prototype does not yet have the failure adaptation capability described in Section 5 and relies on the underlying network layer routing mechanism for failure recovery.

7 Prior and Related Research

To the best of our knowledge, VAN is the first work that presents a complete architectural framework which integrates layer-independent virtualization with active networks to effectively address the challenges we present in the introduction of this proposal although there have been certain research in the area of network virtualization and quite a lot of research in the area of active networks respectively. In this section, we will briefly look at these research and see how they relate to certain aspects of VAN.

Recent advance in computer networks has seen wider use of the concept of virtual network with emerging technologies such as Virtual Local Area Networks (VLANs) [Cisco, 1997 #106][Passmore, 1997 #7], Virtual Private Networks (VPNs) [Scott, 1998 #12], and ATM Virtual Path Connections (VPCs) [Friesen, September, 1996 #93].

A VLAN is a logical subset of a physical LAN that defines a broadcast domain intended to confine LAN broadcast traffic within the boundary of the logical subnet, resulting in better LAN bandwidth utilization and broadcast security. The decoupling of logical subnet and physical net also facilitates the organization and management of the logical subnet based on its role of function.

A VPN is a logical, secure, and private network created on top of physical, insecure, and public network through the combination of several existing technologies such as encryption, tunneling, and firewall. VPN provides a scalable, cost-effective way to create a secure private network for (geographically distributed) organizations whose intra-organization communication must go through insecure public networks.

ATM VPCs are labeled logical paths between two ATM switches called VPC terminators that bundles multiple Virtual Circuit Connects (VCCs) in order to facilitate the setting up, switching, segregating, and managing of VCCs. A collection of VPCs form a logical overlay network that is used in ATM networks to plan for optimal routing and partition of capacity based on traffic pattern.

These technologies virtualize the network at certain protocol layer (e.g., VLAN at layer 2, VPN at layer 3, and ATM VPC at layer 2) and they accomplish specific merits (e.g., VLAN for better bandwidth utilization, VPN for cost-effective security, and ATM VPCs for efficient routing and capacity allocation). In contrast, VAN constructs a virtual data link layer network and allows dynamic programmability of the virtual network services by *applications*. None of these other virtualization technologies has direct relation to dynamic network programmability.

The X-Bone project at ISI [Touch, 1998 #10] is an architecture that facilitates the deployment of “overlay” networks. The “overlay” network is essentially the same as the virtual network in that they are virtual topologies layered on top of physical topology and represents a partitioning of the underlying physical network resources. The difference between X-Bone and VAN again is that X-Bone focuses on the automation of the deployment of overlay networks while VAN, in addition to the provision of virtual networks, also address the dynamic programmability of the deployed virtual networks.

In the area of active networks, a substantial body of work has been done recently.

Early active network research such as ANTS [Wetherall, 1998 #13] and SwitchWare [Smith, 1996 #8] focus on node programmability. They do not address systematic node resource and end-to-end resource allocation and management.

The NetScript project at Columbia [Yemini, 1996 #15] is a language designed specifically for abstracting packet processing primitives to provide universal programmability for network intermediate nodes, similar to the way PostScript abstracts page description primitives to provide universal printing functionality in printers. NetScript and VAN compliment each other in that NetScript provides universal “node” programmability while VAN provides universal “network” programmability. In fact, we mentioned in Section 6 that our prototype works jointly with NetScript Engines.

The Tempest framework [Rooney, October, 1998 #92] is an interesting parallel to VAN, though from the control plane perspective. Its Prospero Switch Divider virtualizes an ATM switch control plane (rather than data plane as in VAN) into what is called switchlets, allowing different control entities to interact with the switch through a standard set of control and management APIs, each confined to a subset of the switch resource. The Tempest also prescribes a service called Network Builder (similar to VAN Builder) which, given a “specification” of a VPN, will interrogate appropriate dividers to create switchlets and instantiate control and management of switch resource necessary for the VPN. So the virtual network aspect of Tempest currently focuses specifically on VPNs rather than general service virtual networks as in VAN.

The Darwin project [Chandra, 1998 #95] conducted at CMU is another example of providing customized control in network via the control plane rather than data plane. Darwin uses so called “delegates” which are code segments that are dynamically dispatched into switches and routers in order to influence the traffic and resource management of these switches and routers according to specific application QoS requirement.

8 Schedule

1. Develop good ways to obtain global topology and resource information. (end of the semester)
2. Develop good heuristics for virtual topology to physical topology mapping. (end of this summer)
3. Develop good formula for computing GPI and analyze conflict resolution protocol. (end of this summer)
4. Study current network reliability techniques such as MPLS fast rerouting, ATM self-healing, etc. to develop good ways of local repair. (end of Fall 2000)
5. Study the feasibility of developing good heuristics for incremental global repair. (end of Fall 2000)

9 Bibliography