1. Virtual Exchange

Author: Petteri Koponen

The VE (Virtual Exchange) module defines classes for control and management operations of a cluster of switches. A cluster consists of a number of interconnected exchanges: the simplest cluster is a single exchange. The VE provides an interface independent of the structure of the underlying cluster - a cluster of exchanges is seen as one (virtual) exchange. **A reader must note that this chaper describes the VE architere and design - the current implementation (described at the end of the chapter) does not support many of the features mentioned.**

1.1 1 Introduction

The VE is planned to consists of a number of "dumb" exchanges, i.e. ATM (Asynchronous Transfer Mode) switches. The switches can be either VTT's (Valtion Teknillinen Tutkimuskeskus) FSRs (Frame Synchronized Ring) or they can come from various vendors. At least in theory, the only requirement of the VE architecture is that the switches implement some open protocol for control and management - this is why they are called "dumb".

In the first phase, a single external workstation controls the switches. The workstation is directly connected to one of the switches with ATM. The workstation must be able to open at least one control and management connection to each of the switches in the VE. These connections are used for e.g. establishing/releasing connections across a single switch and requesting/receiving statistics.

In the VE architecture, the switches only switch cells, i.e. they do not process signalling messages. Also, they hold only a limited amount of information about state of the connections. This is achieved by routing signalling channels from/to external sources to/from the workstation, which handles all the signalling and stores most of the state information of the switches. The signalling messages pass through the switches in the VE transparently, i.e. they are seen as ordinary user data.

An example VE architecture is illustrated in figure 1. The VE consists of four interconnected switches and a workstation, which is called the switch controller. Multiple external terminals and switches are connected to the VE. The switch controller needs to establish control and management connections to switches A - D. When these connections, or at least the relevant part of them, are established, the switch controller commands the switches to set up signalling connections through the VE between itself and the external sources (terminals and switches). These procedures are explained later in detail.

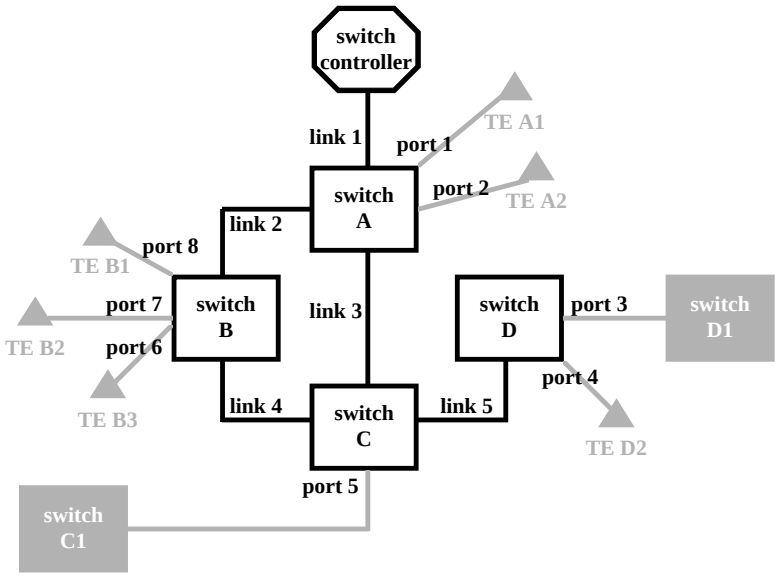

Figure 1 Virtual Exchange

1.2 2 Architecture

This subchapter covers VE's architecture, which has more features than VE's current implementation. Please refer to the last subchapters if you want to know what has been implemented up to now.

1.2.1 2.1 Control and management protocols

As mentioned earlier, the switches must implement some general purprose protocol for external management and control. Currently, two protocols have been considered: VAPI (VTT's API for the FSR) and Ipsilon's GSMP (General Switch Management Protocol).

VAPI. The VTT is implementing an API for the FSR. As opposed to earlier plans, VAPI is going to exist in an external workstation instead of an embedded PC. VAPI consists of functions for establishing, modifying and releasing both point-to-point and point-to-multipoint connections. The workstation is connected to the FSR with ATM. VAPI reservers one or two channels for its use. The FSR's interface card (IC) to which the workstation is controlled differs slightly from the other ICs, but all the ICs could be upgraded, so that the management and control messages would not have to pass through one pre-determined IC. This feature would make the VE more fault tolerant (this will be discussed later).

The current VAPI implementation runs on Linux. It uses Linux's socket-based ATM API to pass AAL-0 (ATM Adaptation Layer) PDUs (Protocol Data Unit) between the workstation and the FSR. VAPI uses AAL-0 PDUs as its control messages and the messages may pass through several FSRs, which may naturally take some time or the messages may be lost. These losses are not detected, because currently the VAPI functions do not wait for confirmation from the FSR, but return immediately after

sending the AAL-0 command message to the switch. VAPI is planned to be upgraded to support confirmations, but at least the first version does not do that.

GSMP. Developed by Ipsilon, GSMP has quite much the some functionality as VAPI has. It has been described in detail in RFC 1987, which has been released in August 1996. It seems that GSMP is gaining wide industrial acceptance and it has already been implemented in e.g. some of Hitachi's and Digital's ATM switches. GSMP does not define an interface as VAPI does - instead, it is a simple protocol with a few PDUs and a state machine. In principle, the protocol consists of request and corresponding responses. Also, the switch can inform the workstation of asynchronous events such as a link going down. The GSMP messages are variable length and are encapsulated directly in an AAL-5 PDUs with an LLC/SNAP header. The default virtual channel for LLC/SNAP encapsulated messages is VPI = 0, VCI = 15. GSMP can be used to control a number of switches by instantiating it multiple times.

It is not known, what kind of a protocol VAPI uses to pass messages between the workstation and the FSR. Because VAPI's and GSMP's basic features are very similar, it should be considered, if VAPI could use native GSMP or its enhanced superset to control the switch. This would be commercially a very sound solution - the FSR could be used as an IP switch with Ipsilon's IP software or the needed software could be implemented by some other organization.

1.2.2 2.2 Control and management connections

To control a switch with either VAPI or GSMP, the switch controller must be able to send/receive AAL-0/AAL-5 PDUs to/from it. The case is very simple when the controller is connected to the switch with a physical link: the controller and the switch must send/receive PDUs to/from a pre-determined channel(s) (VAPI's control channel(s) or GSMP's VPI = 0, VCI = 15). If the VE consists of multiple switches, the case becomes more complicated: the controller has to set up the connections in stages. In figure 1, if the controller wants to build a control and management connection to switch B, if first has to ask switch a (using the pre-determined channel(s)) to establish a VPI/VCI pair between itself and switch A, and then to connect this connection to switch B's pre-determined control channel. The switch C could then be connected either via switch B or switch A (and after this, switch D via switch C). The biggest problem with this scheme comes from the bidirectionality of the control channels - how can e.g. switch A know that PDUs coming from switch B's control channel are not ment to itself, but must be routed to the switch controller. This problem does not exist if there is only one intelligent IC in a FSR, but in this case, only one route can be used to establish a control channel to a switch, which makes the system more sensitive to link failures. This problem must be discussed with the VTT.

In case that multiple control connections could be established between the controller and a single switch, procedures similar to MTP3's (Message Transfer Part 3) changeover and changeback procedures could be applied, i.e. the control messages could be re-routed in case of link failures etc. For example, if a primary control connection between the

controller and switch C would pass through switch A and the secondary connection through switches A and B, failure of switch B would automatically activate the secondary and deactivate the primary connection.

According to latest news from the VTT, a native FSR cluster (cluster consisting solely of FSRs) could be controlled through VAPI without establishing the control channels. This would be achieved by using FSR's ring addresses in addition to normal port addresses in the control messages. A ring represents here one FSR switching fabric. This means that also distant switches could be controlled with simple API function calls. This feature has not been implemented in the FSR (or VAPI) until now.

1.2.3 2.3 Signalling

The VE can accept signalling from external switches and terminals when the needed control and signalling connections have been set up. The signalling connections are established through possibly multiple switches between the external sources and the switch controller. For example, when the controller wants to set up a signalling connection to switch C1 through switch B, it orders switch A to connect some VPI/VCI pair in link 1 to another pair in link 2. Then it orders switch B to connect the previous VPI/VCI pair in link 2 to some pair in link 4. Finally, it commands switch C to connect the previous pair in link 4 to the default signalling channel (VPI = 0, VCI = 5) in port 5, and the signalling connection is fully established. The connection can be released in reverse order.

The signalling messages are encapsulated in AAL-5 CPCS PDUs. The NIC (Network Interface Card) of the Linux-based switch controller can handle the necessary segmentation and reassembly functions, which makes the implementation more efficient and spares resources of the signalling software. The switch controller is not limited to map only signalling connections - it can connect to any any VPI/VCI pair of any exterior port of the VE (ports 1 - 8 in figure 1). This allows e.g. data connections or GSMP connections to be established between the controller and some external source.

1.3 3 Implementation details

One of the goals of the VE architecture is that signalling software is independent of the topology of the cluster. Due to this, the VE interface must be very general, which propably makes the implementation rather complex. As mentioned in the beginning, user of the VE module sees the VE as a single ATM switch with a great number of ports (ports 1 - 8 in the simplified example of figure 1). This implies that VE's internal control and signalling connections and resources (like available bandwidth of links or switches) must be managed under the VE interface, i.e. transparently to the ATM signalling and management protocols.

The VE interface is similar to VAPI or GSMP. It provides basic functions for establishing, modifying and releasing connections (point-to-point or point-to-multipoint) and for different management and configuration operations like configuring ports or

requesting status of a switch. The biggest problem is that these operations take a variable amount of time, especially when applied to a switch on the edge of a large cluster - the tasks must either run on a process/thread of their own (and block while waiting for the response) or the functions must be implemented by using state machines (which go to a waiting state when the request is sent and perhaps use timers to notify if the request has failed). This problem is partially solved when using VAPI's current version, which does not support confirmations: at least the time-critical connection management functions do not block. However, the VE architecture has been designed to be asynchronous meaning that it supports asynchronous notifications of successful or failed operations.

The VE interface controls the switches of the cluster through the Fabric interface. The VE uses Fabric objects to establish/modify/release physical, real connections (as opposed to logical connections accross the cluster) accross individual switches, to ask status of one switch, etc. This version of the paper does not discuss much how the VE internally establishes the connections. The VE must order each switch along the connections route to establish a connection. It also has to check if the connection setup of each switch was successful and either to return a success indicator or to notify the client with some other means. This procedure requires further study.

The VE interface will be used by several clients, e.g. Call Control and SAAL (Signalling AAL). It provides an interface of a single switching fabric. Clients can e.g. create connections (`veVE::EstablishVC() method`) accross this (virtual) fabric. They can also request status information from the VE, configure its outer ports, etc. The client does not see concrete fabric objects, connections or interior ports. Interior ports mean here ports that connect switches of the cluster to each other. Outerior ports connect the cluster to workstations and switches outside the cluster. What the clients see are `veVE` object (implemented according to Singleton design pattern), `veVCProxy` objects it has created and the outerior ports of the cluster (or the VE), i.e. `vePortProxy` objects. The VE module is designed so that clients only need to include `ve.h` file that contains veVE class definition.

1.4 4 Features implemented

The implementation documented here is minimal and supports the smallest set of functions that were needed in the first prototype of the TOVE switch. Features include:

- Support for **a single** FSR or Back-to-Back fabric. These simplifies the operations substantially but does not preclude future extensions.

- Setup/teardown of **point-to-point VCs**. No VPs or point-to-multipoint VCs are supported, although these features could be very easily added in the VE (the current VAPI supports them, too).

- Automatic allocation (when required) mechanism for VPI/VCI pairs.

1.4.1 4.1 How to use the VE?

As mentioned before, `ve.h` is the only class the user needs to include to use the VE. All the called methods belong to the `veVE` class or the classes (actually objects) it returns. The first thing to do is always the initialization.

- **Initializing the VE**. The VE is given an object that contains a reference to the configuration file. The VE delegates interperation of the configuration information to individual fabrics. See `switch.conf` file for details about the configuration options. **The initialization must be done only once!**

- **Obtaining proxies to ports**. Ports must be referred by proxies to them. The clients can not access the port objects directly.

- **Establishing a point-to-point VC**. The current version of the VE requires incoming and outgoing points (port/VPI/VCI combinations) and an acknowledgement object as parameters. An acknowledgement object is ment for the future asynchronous use of the VE (e.g. when GSMP has been deployed). The `success()` or `failure()` method of the object is called when the VC establishment has succeeded or failed, respectively. The client is not supposed to operate immediately returned `veVCProxy` object before either of these methods has been called.

- **Releasing the VC**. All the VCs must be released when they are not needed anymore so that the reserved resources are released.

Below is an example of the above phases.

```
// Prepares the configuration file for reading.
tvFileParser configs("../switch.conf");

// veVE implements Singleton design pattern. This returns a
// pointer to the only instance of the veVE class.
veVE *instanceOfVE = veVE::instance();

// Initializes the VE.
if (instanceOfVE->initialize(configs) != veOK_ERR)
{
    // The VE was not initialized properly.
    Handle error and exit (or do something else).
}

// Gets proxies to ports.
vePortProxy port1, port2;
if ((getPortProxy(port1, 1) != veOK_ERR) ||
    (getPortProxy(port2, 2) != veOK_ERR))
{
    // Some of the ports did not exist.
    Handle error and exit.
}

// Creates two points and lets the VE to allocate VPIs and
// VCIs in the ports.
vePoint inPoint(port1, veUNDEFINED_VPI, veUNDEFINED_VCI);
```

```
vePoint outPoint(port2, veUNDEFINED_VPI, veUNDEFINED_VCI);

// Establishes a connection. MyAckObj is derived from veAckObj
// => it implements success() and failure() methods.
xxMyAckObj ackObj;
veVCProxy vcProxy = instanceOfVE->establishVC(inPoint,
                                              outPoint,
                                              ackObj);

... program waits for a call to ackObj's success() or failure()
    methods and does something with the established VC after
    that ...

// Finally, the VC is released.
vcProxy.release();
```

1.5 5 Known bugs and flaws

The biggest flaws are marked with ++TODO++ in the code. These places (and many others) will be changed/extended in the next release of the VE module.

- **The VPI/VCI allocation mechanism is currently just for testing purproses!** In real use, the VPI/VCI space would run out.
- The error handling is very primitive.
- The module supports only a small portion of VAPI's and especially FSR hardware's features.

1.6 6 Future development

Most of the VE functionality is going to be implemented during winter/spring -97. All the VE components (classes) are under development and are going to substantially extended. These include:

- Support to point-to-multipoint VCs. This is fairly easy to implement at the VE level, but much harder in Call Control and other high-level modules.
- More advanced tracing and logging features. The classes should support methods and mechanisms for status enquires and they also should be able to return/log some statistic about their activities. Currently, mechanisms like these are not very useful due to absence of messages, i.e. notifications and confirmations from FSR hardware to the switch controller.
- Implementation of a GSMP fabric and related classes. It seems that the VTT may provide GSMP support in the FSR hardware, which would benefit also the the TOVE project, mainly because GSMP has become almost a de facto standard in distributed control of ATM switching hardware.

1.7 7 Statistics

The statistics below do not include basic research on ATM and B-ISDN technologies etc. The VE module does not contain much code - most of the work was done in studying the existing APIs and protocols for switch management and also in designing and evaluating the VE architecture, i.e., is it possible and feasible to manage multiple switches by a single switch controller.

Activity	Research	Design	Coding	Reviews	TOTAL
Duration (h)	80	40	60	20	200

Table 1 Duration of activities

Lines Of Code (LOC)	Number of files	Number of classes
1880	25	27

Table 2 Metrics

1.8 8 References

Lazar, A., Marconcini, F., *Towards Open API for ATM Switches*, Center for Telecommunications Research, Columbia University, New York, March 1996, http://www.ctr.columbia.edu/.

Newman, P., Edwards, W., Hinden, R., Howman, E., Liaw, F., Minshall, T. G., *Ipsilon's Generic Switch Management Protocol Specification version 1.1*, Ipsilon Networks, Palo Alto, August 1996, http://www.ipsilon.com/.

Haatanen, T., Tuliluoto, J., *FSR switch Application Programming Interface*, VTT Information Technology, Espoo, October 1996, http://www.vtt.fi/tte/.