CrossBow: Solaris Network Virtualization & Resource Control

1. CrossBow (the name):

It makes some sense to explain the relatonship between the technology (Network Virtualization and Resource Control) and the project name (CrossBow). It is believed that Crossbow was invented in 341B.C. in China but the use became prevalent in middle ages specially when steel was used to make the weapon. More powerful Crossbows could penetrate the armour at 200 yards and gave the typical horse mounted knights real nightmares. But the biggest differentiator was the simplicity in their use. Crossbow could be used effectively after a week of training, while a comparable single-shot skill with a longbow could take years of practice.

Similary, if you take a look at the existing QOS mechanisms on a end host, they are very difficult to use and normally take a very skilled administrator to use effectively. Even then, the existing QOS mechanism come with heavy performance penalties which is also pretty common with any kind of virtualization as well. In Solaris land, we have invented a new way of imposing bandwidth resource control as attribute to a real or a virtual NIC such that it is built in as part of the Solaris network stack and comes without any performance penalties. Since the virtualization aspects and/or resource control aspects are just the attributes of the NIC/VNIC (specified when a NIC or Virtual NIC is created), a normal user and configure them without needing a docterate in QOS or virtualization. "CrossBow" was the most suitable name for this project since we are trying to achieve similar results in the field of virtualization and resource control as the weapon did in medivial times in the battlefield.

2. CrossBow (the background):

Crossbow provides the building blocks for network virtualization and resource control by creating virtual stacks around any service (HTTP, HTTPS, FTP, NFS, etc.), protocol (TCP, UDP, SCTP, etc.), or Virtual machines like Containers, Xen and ldoms.

The project allows the system administrator to carve out any physical NIC into multiple virtual NICs which are pretty similar to real NICs and are administered just like real NICs. Each Virtual NIC can be assigned its own priority and band-width on a shared NIC without causing any performance degradation. The virtual NICs can have their own NIC hardware resources (Rx/Tx rings, DMA channels), MAC addresses, kernel threads and queues which are private to the VNIC and are not shared accross all traffic. In case of Solaris Containers, the Container can be assigned a virtual Stack Instance as well along with one or more virtual NICs. As such traffic for one VNIC can be totally isolated from other traffic and assigned any kind of limits or guarantees on amount of bandwidth it can use.

3. Overview:

Project Crossbow extends Solaris reach in several markets.

3a. OS/Network/Server Consolidation:

The application, network and server consolidation environments where both OS and network virtualization play a big role. This market is typically driven by the cost of owning and managing physical machines and physical networks. The sweet spot for these horizontally scaled environment are the 2-4 socket machines which appear as 4-8 CPU machines in case of x86/x64 systems and 32-64 CPU machines in case of SUN's new Niagara based servers. From total cost of ownership perspective, these blades have only one physical NIC (1Gb or 10Gb) but are trying to run multiple virtual machines (Xen, Containers, ldoms) which have to share the NIC resources and the available bandwidth.

The problem gets worse because for 3 decades we have been designing application to go as fast as possible and any congestion control is the job of the transport layer (if at all). So if one virtual machine is using UDP based traffic, then other virtual machines on the same system using TCP traffic will suffer badly. Even within same transport (TCP for instance), bulk througput applications like ftp/http etc will have a very negetive impact on interactive traffic and latency sensitive applications.

The goal of the project Crossbow is to different virtual machines share the common NIC in a fair manner and allow system administrators to set preferential policies where necessary (e.g. the ISP selling limited B/W on a common pipe) without any performance impact.

3b. Traditional QOS and application consolidation:

Exisiting host based QOS mechanism are very complex to setup and typically come with a sizable performance penalty and increase in latency. The big part of the problem is the interrupt based delivery mechanism for inbound packets and the QOS being implemented by a separate layer (typically between NIC driver and IP). The network and transport layer of the host stack is unware about the QOS layer. The packets are already delivered to the host memory by means of interrupts and the QOS layer needs to classify the packets to various queues before it can apply the policies. In case the packet can not be processed because the bandwidth usage for that class is exceeded, it sits in a queue while still consuming system memory.

Project Crossbow integrates stack virtualization and QOS as part of the stack architecture itself to offer a large subset of QOS type functionality at zero performance penalty and simple administrative interfaces. It also integrates diffserv with the stack where a virtual NIC can set and read the diffserv based labels. Since Crossbow architecture is limited in differentiating the traffic based on layer 2, 3, and 4 headers only i.e. the VLAN tag, local mac address, local IP address, protocol, and ports; the functionality offered is a subset of exisiting QOS mechanism although it covers 90% of the use cases without any performance penalty. This is the prime reason why project Crossbow refers to the bandwidth related policies as 'Bandwidth resource control' instead of QOS.

3c. Horizontally scaled markets:

This is the market segment made up of low priced volume servers (typically 2-4 socket machines) which offer services which require little or no sharing of data between them. The small servers can be standalone machines in a rack or blades in a chasis. Grids are another way to use volume servers to achieve the output of the traditional large SMP machines or main frames.

In case of blades which share a common 10Gb NIC to the chasis, Crossbow again provides the sharing of bandwidth in a fair manner. In addition, the Crossbow provided APIs for network management, virtualization and bandwidth resource control can be use by 3rd party management softwares to propogate the common policy throughout the server farm or all the blades in the chasis. In a Solaris based homogenous environments, its very easy to mark an application or a virtual machine (based on port or IP address) as critical and propogate the same policy through all the machines. The diffserv labels can be added appropriately such that the policy is honoured by all machines and network element in the center.

4. Technical problems in exisiting architectures:

As mentioned earlier, the host based QOS systems work as a layer between the network stack and as such are pretty inefficient in providing the QOS services required of them. But that is not all.

The exisiting interrupt driven packet delivery model pecludes any kind of policy enforcement and fair sharing. When a NIC interrupt is raise, it is at a highest priority and the CPU has to context switch whatever processing to deal with the interrupt. Most of the time, the processing of a critical packet is interrupted to deal with the arrival of a non critical packet.

The anonymous packet processing in the kernel is another major problem in virtualizing the stack and enforcing any kind of bandwidth resource control (including fairness). 80% of the work is already done for an incoming packet when the stack determines that no one is actually interested in the packet and it needs to drop it. In other words, the cost of dropping unwanted packets is too high.

Everything in the host flows through common queues and is processed by common threads which make enforcing policies based on traffic type very difficult. Recv or xmit of each packet impacts processing on any other packet on that particular CPU.

In most of the virtualized environments, the pseudo NIC in the virtual machines has no way of knowing about the hardware capabilities of the real hardware (even simple things like hardware checksum) because of the presense of the bridge in between and ends up making negetive performance impact. In addition, there is no mechanism to share the NIC in a fair manner. The transition of typical packet from the dom0 to domU also causes severe performance problems.

5. CrossBow Architecture:

The Crossbow architecture starts out by integrating network virtualization and resource control as part of the stack architecture. The Solaris 10 network stack has already been designed for the next decade where the connection to CPU affinity is maintained and the upper stack has tight control over the NIC resources.

Crossbow builds on top of that by pushing the classification of packets based on services, protocols or virtual machines as far below as possible. If the NIC hardware itself has ability to divide onboard memory into segements/queues (know as Rx and Tx rings) which can preferably haev their own DMA channels and MSI-X interrupts, the stack programs the NIC classifier to classify packets based on configured policies to different Rx rings. Each Rx/Tx ring is owned by a CPU and a separate kernel queue know as serialization queue which controls the rate of packet arrival into the system based on configured bandwidth.

The Rx/Tx ring, the associated DMA channel, MSI-X interrupt, the serialization queue, the CPU, and processing threads are all unique for the service, protocol or virtual machine in question and can be assigned a unique MAC address and a Virtual NIC which becomes the administration entity that can be administered like a normal NIC. The NIC classifier drives the incoming packets to the correct RX ring from where the Squeue owning the Rx ring (and VNIC) will pull the packets via polling mode based on fair sharing of resources or configured bandwidth. The interrupt mode is used only when the Squeue has no packets to process and the Rx ring is empty. Each individual Rx ring is dynamically switched between interrupt and polling mode. Incoming packets that exceed the configured bandwidth limit remain in the NIC itself in their corresponding Rx ring and are pulled in the system only when they are ready to be processed.

The creation of an administrative entity (VNIC) is optional and typically associated with a virtual machine like Solaris containers, Xen or ldoms. For application or protocol based resource control, a separate data path is created to provide the isolation and resource control but a VNIC is not configured.

As mentioned above the VNIC is just an administrative entity. If the classification has already been done by the NIC to a particular Rx ring, the packets as delivered directly to IP layer by means of function calls when Rx ring is interrupt mode or the squeue residing in IP layer pulls the packet chain directly from the Rx ring when in the polling mode. In essence, the entire data link layer is bypassed resulting in improved performance and lower latencies. If the VNIC is placed in promiscous mode, the data link bypass is abandoned and the Rx ring delivers packets via the VNIC layer which creates a copy of the packet for promiscous stream. Similarly, in polling mode, the squeues poll entry point are changed to point at VNIC which is turns pulls the packets from Rx rings, makes a copy and then gives the chain to the Squeue poll thread.

The entire layered architecture is built on function pointers know as 'upcall_func' and 'downcall_func' with corresponding 'upcall_arg' and 'downcall_arg' for context. Every layer provides a pointer of its recv function as 'upcall_func' and a context as 'upcall_arg' to the layer below. Similarly, every layer provides pointer to its transmit function as 'downcall_func' and a context cookie as 'downcall_arg' to layer above. This is how the packet path is constructed. Any layer can short circuit itself out by providing the 'upcall_func' and 'upcall_arg' of the layer above to layer below (and same for transmit side if needed). All context cookies for a layer work on reference based system when each layer pointed to it gets a reference and ensure that data structures don't get freed till all references are dropped.

In case, the NIC hardware does not have classification capability (unlikely since most of intel, broadcom and SUN 1Gb NICs and pretty much all 10Gb NICs shipping for past several years have this capability) or have run out of the classification capability, the architecture provides a classification capability in the mac layer and employs soft rings which are similar to functionality as NIC hardware classifier and RX rings. The NIC hardware layer coupled with lower MAC layer and soft rings are termed as 'Pseudo Hardware layer'. A request by administartor to create a new VNIC or flow will always return successful from the pseudo hardware layer. The pseudo hardware layer manages the hardware and software classification capability and Rx rings and soft rings transparently from upper layers.

6. Crossbow layers, data structures and packet flow:

Its easier to illustrate this with 2 flows. The first one is for IP_addr = a.b.c.d && TCP and it goes through normal path via Upper dls etc. This is under the assumption that either snoop (or someone else in DLS) is interested in this flow and we can't bypass data link processing. The squeue poll function in this case is dls_poll_ring and argument is dls_impl_t.

The 2nd flow is for IP_addr = m.n.o.p && port = 80 && TCP which is unique and no one is interested in snooping it. In this case, the dls layer allows itself to be pypassed by setting the upcall_func and upcall_arg for soft_ring/Rx_rings to directly call into IP. The squeue is directly polling the H/W Rx ring in this case.

```
ill_rx_ring =   | ill_rx_ring1  |       |  ill_rx_ring2 |       |
ill_poll_cookie=|dls_flow_impl  |       |rx_ring2_cookie|       |
ill_poll_func = |dls_poll_ring  |       |bge_poll_ring  |       |
        |       |               |       |               |       |
        |       |               |       |               |       |   Squeues
ill_sr_cookie = |sr_ring1_cookie|       |sr_ring2_cookie|       |
        |       |               |       |               |       |
        |       +---------------+       +---------------+       |
        |       Squeue 1 (CPU 1)        Squeue 2 (CPU 2)        |
        |                                                       |
        |                                                       |
        ---------------------------------------------------------
                                                                IP
_______________________________ ip_input __________________________________

        ---------------------------------------------------------
        |       +---------------+       +---------------+       |
        |       |               |       |               |       |
dls_upcall_func=| ip_input      |       |  ip_input     |       |
sr_upcall_arg1 =| ill           |       |  ill          |       |
sr_upcall_arg2 =| ill_rx_ring1  |       |  ill_rx_ring2 |       |
        |       |               |       |               |       |
dls_poll_func = | bge_poll_ring |       |  NULL         |       |
dls_poll_arg =  |rx_ring1_cookie|       |  NULL         |       |
        |       |               |       |               |       |
sr_rx_cookie =  |sr_ring1_cookie|       |sr_ring2_cookie|       |
        |       |               |       |               |       |
        |       +---------------+       +---------------+       |
        |       dls_flow_impl1          dls_flow_impl2          |    Upper
        |                                                       |    DLS
        |                       dls_impl                        |
        ---------------------------------------------------------

_______________________________ vnic_rx ___________________________________

                                                                Pseudo H/W
        ---------------------------------------------------------
        |       +---------------+       +---------------+       |
        |       |               |       |               |       |
sr_upcall_func =| i_dls_link_rx |       |  ip_input     |       |
sr_upcall_arg1 =| dls_flow_impl |       |  ill          |       |
sr_upcall_arg2 =| ill_rx_ring1  |       |  ill_rx_ring2 |       |
        |       |               |       |               |       |
        |       |               |       |               |       |
sr_rx_cookie =  |rx_ring1_cookie|       |rx_ring2_cookie|       |
        |       |               |       |               |       |
        |       +---------------+       +---------------+       |    Nemo
        |       Soft Ring 1             Soft Ring 2             |    Lower
        |       (sr_ring1_cookie)       (sr_ring2_cookie)       |    MAC
        |                                                       |
        |                                                       |
        |       -----------------------------------------       |
        |       | IP_addr = a.b.c.d && TCP      |  SR1  |       |
        |       | IP_addr = m.n.o.p && port = 80|  SR2  |       |
        |       -----------------------------------------       |
        |               S/W FLOW CLASSIER                       |
        ---------------------------------------------------------
_______________________________ mac_rx ____________________________________

                                                                Device Driver
____________________________________________________________________________

                                                                Real H/W
        ---------------------------------------------------------
rx_upcall_func = mac_rx                         ip_input        |
rx_upcall_arg1 = dls_flow_impl1                 ill             |
rx_upcall_arg2 = ill_rx_ring1                   ill_rx_ring2    |
        |       +-------+                       +-------+       |
        |       |       |                       |       |       |
        |       |       |                       |       |       |
        |       |       |                       |       |       |
        |       +-------+                       +-------+       |
        |       Rx Ring 1                       Rx Ring 2       |    NIC
        |       (RR1)                           (RR2)           |
        |                                                       |
        |                                                       |
        |       -----------------------------------------       |
        |       | IP_addr = a.b.c.d && TCP      |  RR1  |       |
        |       | IP_addr = m.n.o.p && port = 80|  RR2  |       |
        |       -----------------------------------------       |
        |               H/W FLOW CLASSIER                       |
```

7. The administrative model:

Crossbow introduces a new command called 'netrcm' and further augments 'dladm' which was introduced as part of the new high performance device driver framework (GLDv3) in Solaris 10.

'dladm (1M)' - This is primarily used to create, modify and destroy VNIC based on mac or IP addresses. The created VNIC is visible and managed by ifconfig just like any otehr NIC and can get its IP address assigned via DHCP if necessary.

The examples below can illustrate this better:

```
     Example 1: Configuring VNICs

     To create two VNICs interfaces with vinc-ids 1 and 2
     over a single physical device bge0, enter the following com-
     mands:

     # dladm create-vnic -d bge0 1
     # dladm create-vnic -d bge0 2
     The new links will be called vnic1 and vnic2.

     Example 2: Configuring VNICs and allocating bandwidth & priority


     To create two VNIC interfaces with vinc-ids 1 and 2
     over a single physical device bge0 and make vnic1 a higher
     priority VNIC using factory assigned MAC address with guarantee
     to use upto 90% of the bandwidth and vnic2 having a lower priority
     with a random MAC address and a hard limit of 100Mbps:

     # dladm create-vnic -d bge0 -m factory -b 90% -G -p high 1
     # dladm create-vnic -d bge0 -m random -b 100M -L -p low 2

     Example 3: Configure a VNIC by choosing a factory MAC address

     To create a VNIC interface with vinc-id 1 by first
     listing the factory available MAC address and then using one
     of them:

     # dladm show-dev -d bge0 -m
     bge0
             link: up        speed: 1000   Mbps       duplex: full
     MAC addresses:
slot-ident      Address                 In Use
1               0:e0:81:27:d4:47        Yes
2               8:0:20:fe:4e:a5         No

     # dladm create-vnic -d bge0 -m factory -n 2 1

     # dladm show-dev -d bge0
     bge0
             link: up        speed: 1000   Mbps       duplex: full
     MAC addresses:
slot-ident      Address                 In Use
1               0:e0:81:27:d4:47        Yes
2               8:0:20:fe:4e:a5         Yes
```

```
     Example 4: Configuring VNICs sharing a MAC address

     To create two VNICs with vnic-id 1 and 2 by first listing the
     available factory assigned MAC addresses and then picking one
     that will be shared by the newly created VNICs

     # dladm show-dev -d bge0 -m
     bge0
            link: up        speed: 1000   Mbps       duplex: full
     MAC addresses:
slot-ident      Address                 In Use
1               0:e0:81:27:d4:47        Yes
2               8:0:20:fe:4e:a5         No

     # dladm create-vnic -d bge0 -m shared -n 2 1
     # dladm create-vnic -d bge0 -m shared -n 2 2

     Example 5: Creating a VNIC with user specified MAC address

     To create a VNIC with vnic-id 1 by providing a user specified
     mac address

     # dladm create-vnic -d bge0 -m 8:0:20:fe:4e:b8
```

'netrcm (1M)' - This command is primarily used to provide isolation and private resources to an application traffic or protocol. In addition, we can also configure bandwidth limits and guarantees for the flows. Again some example can illustrate the usage better:

```
     Example 1: Create a policy around mission critical port 443 traffic
     which is https service.

     To create a policy around inbound https traffic on a https server
     so that https gets it dedicated NIC hardware and kernel TCP/IP
     resources. The policy-id specified is https-1 which is used to
     later modify of delete the policy.

     # netrcm add-policy -d bge0 -H transport = TCP local port = 443 https-1

     Example 2: Modify an existing policy to add bandwidth resource control

     To modify https-1 policy to add bandwidth control and give it a
     high priority

     # netrcm modify-policy -d bge0 -b 90% -G -p high https-1

     Example 3: Limit the bandwidth usage of UDP protocol

     To create a policy for UDP protocol so that it can not consume more
     than 10% of available bandwidth. The policy-id is called limit-udp-1.

     # netrcm add-policy -d bge0 -b 90% -L -p low limit-udp-1
```

8. Crossbow Observability - Stats, history and APIs:

Apart from the functionality related to network virtualization and bandwidth resource control, Crossbow offers a whole range of news tools and mechanism to understand the bandwidth usage. Administrators can see real time bandwidth usage for various VNICs or configured flows (via 'netrcm') without causing any performance penalties.

The Rx rings and squeues dealing with a particular flow keep track of normal stats which are pulled by a userland daemon from time to time. The daemon also logs the information in special log files which allows users to see history at any given time. A user can request usage for a time period in past to understand the system behaviour.

Crossbow will provide more tools to help capacity planning by allowing the system to be put under capacity planning mode where bandwdith usage for top traffic is monitored and displayed.

All the observability and administrative interfaces can be accessed by APIs which allow other applications to use and manage the system.

9. Resources:

Crossbow project page on OpenSolaris is a good source of information http://www.opensolaris.org/os/project/crossbow

The Crossbow mailing list is where all the day to day business for the project is conducted. Anyone can join the mailing list crossbow-discuss@opensolaris.org.

Crossbow slide presentation can be found here Crossbow Team members are:

```
    * Kais Belgaied
    * Stephanie Brucker
    * Eric Cheng
    * Nicolas Droux
    * Markus Flierl
    * Carol Gayo
    * Mohan Iyer
    * Darrin Johnson
    * Michael Lim
    * Rajagopal Kunhappan
    * Erik Nordmark
    * Ethan Solomita
    * Thirumalai Srinivasan
    * Sunay Tripathi
    * Nicky Veitch
    * Bill Watson
    * Roamer Lu
```

Email: first.last@sun.com