Pcap Filter Syntax Reference

Qosium uses Pcap syntax for defining manual packet filters. To master at least the very basic filtering cases is very useful when working with Qosium. This section provides information about general Pcap filter syntax that is often relevant to Qosium.

Table of Contents

1. Pcap Filter Syntax ... 3
1.1. Primitives ... 3
1.1.1. Host ... 3
1.1.2. Ether ... 3
1.1.3. Net & Mask ... 3
1.1.4. Port ... 4
1.1.5. Packet Size ... 5
1.1.6. Protocol ... 5
1.1.7. Broadcast ... 6
1.1.8. WLAN ... 6
1.1.9. VLAN ... 7
1.1.10. Multiprotocol Label Switching (MPLS) ... 7
1.1.11. Point-to-Point Protocol over Ethernet (PPPoE) ... 8
1.2. Relation Expression ... 8
1.3. Combining Primitives ... 9
2. Examples ... 9
2.1. Basic and Common Scenarios ... 9
2.2. PROFINET over Ethernet ... 10
2.3. Odd and Even Port Numbers ... 11
3. Special Cases ... 11
3.1. Sometimes the Order of Primitives Matter ... 11
3.2. Curious Filtering Cases ... 11
4. Glossary ... 13

1. Pcap Filter Syntax

While some information is given next, a full syntax reference is found here.

1.1. Primitives

Important Pcap filter primitives are the following:

1.1.1. Host

Syntax	Condition
`dst host <host>`	IPv4/v6 destination field of the packet is `host`, which may be either an address or a name
`src host <host>`	IPv4/v6 source field of the packet is `host`
`host <host>`	IPv4/v6 source or destination of the packet is `host`

Any of the above host expressions can be prepended with the keywords `ip`, `arp`, `rarp`, or `ip6`, as in `ip host <host>`, which is equivalent to `ether proto \ip and host host`. If the host is a name with multiple IP addresses, each address will be checked for a match.

1.1.2. Ether

Syntax	Condition
`ether dst <ehost>`	Ethernet destination address is `ehost`, which may be either a name from /etc/ethers or a number
`ether src <ehost>`	Ethernet source address is `ehost`
`ether host <ehost>`	Ethernet source or destination address is `ehost`

True if the packet used the host as a gateway. I.e., the Ethernet source or destination address was host, but neither the IP source nor the IP destination was the host. The host must be a name and must be found both by the machine's host-name-to-IP-address resolution mechanisms (hostname file, DNS, NIS, etc.) and by the machine's host-name-to-Ethernet-address resolution mechanism (/etc/ethers, etc.). (An equivalent expression is ether host ehost and not host host, which can be used with either names or numbers for host/ehost.) This syntax does not work in an IPv6-enabled configuration at this moment.

```
gateway <host>
```

1.1.3. Net & Mask

Syntax	Condition
`dst net <net>`	IPv4/v6 destination address of the packet has a network number of `net`

Syntax	Condition
`src net <net>`	IPv4/v6 source address of the packet has a network number of `net`
`net <net>`	IPv4/v6 source or destination address of the packet has a network number of `net`
`net <net>mask <netmask>`	IPv4 address matches net with the specific netmask. May be qualified with `src` or `dst`. Notice that this syntax is not valid for IPv6 net
`net <net>/<len>`	IPv4/v6 address matches `net`with a netmask `len` bits wide. May be qualified with `src` or `dst`

Net may be either a name from the network's database (/etc/networks, etc.) or a network number. An IPv4 network number can be written as a dotted quad (e.g., 192.168.1.0), dotted triple (e.g., 192.168.1), dotted pair (e.g., 172.16), or a single number (e.g., 10); the netmask is 255.255.255.255 for a dotted quad (which means that it's really a host match), 255.255.255.0 for a dotted triple, 255.255.0.0 for a dotted pair, or 255.0.0.0 for a single number. For IPv6 addresses, the network can only be defined by using the network number and the mask length. Thus, for example, filter

```
net fe80:1234:5678:9abc:0000:0000:0000:0000/64
```

includes all traffic containing the IPv6 addresses is in the range: fe80:1234:5678:9abc:0000:0000:0000:0000 - fe80:1234:5678:9abc:ffff:ffff:ffff:ffff.

1.1.4. Port

Syntax	Condition
`dst port <port>`	Packet has a destination port value of `port`
`src port <port>`	Packet has a source port value of `port`
`port <port>`	Either the source or destination port of the packet is `port`
`dst portrange <port1>-<port2>`	Packet has a destination port value between `port1` and `port2`
`src portrange <port1>-<port2>`	Packet has a source port value between `port1` and `port2`
`portrange <port1>-<port2>`	Packet has a source or a destination port value between `port1` and `port2`

True if the packet is IPv4/IPv6 TCP, IPv4/IPv6 UDP, or IPv4/IPv6 SCTP, in some systems, and has a destination port value of port. The port can be a number or a name used in /etc/services. If a name is used, both the port number and protocol are checked. If a number or ambiguous name is used, only the port number is checked (e.g., dst port 513 will print both TCP/login traffic and UDP/who traffic, and port domain will print both TCP/domain and UDP/domain traffic).

Any of the above port or port range expressions can be prepended with the keywords `tcp or`udp`. For example,`tcp src port `matches only TCP packets whose source port is`.

1.1.5. Packet Size

Syntax	Condition
`less <length>`	Packet has a length less than or equal to `<length>`. This is equivalent to `len <= <length>`
`greater <length>`	Packet has a length greater than or equal to `<length>`. This is equivalent to `len >= <length>`

1.1.6. Protocol

Syntax	Condition
`tcp`	Short for `proto tcp`
`udp`	Short for `proto udp`
`icmp`	Short for `proto icmp`
`ip proto <protocol>`	Packet is an IPv4 packet of protocol type `<protocol>`
`ip protochain <protocol>`	Packet is IPv4 packet, and contains protocol header with type `<protocol>` in its protocol header chain
`ip6 proto <protocol>`	Packet is an IPv6 packet of protocol type `<protocol>`
`ip6 protochain <protocol>`	Packet is IPv6 packet, and contains protocol header with type `<protocol>` in its protocol header chain
`ether proto <protocol>`	Packet is of ether type `<protocol>`, where protocol can be `ip`, `ip6`, `arp`, `rarp`, `atalk`, `aarp`, `decnet`, `iso`, `stp`, `ipx`, `netbeui`, `lat`, `moprc`, `mopdl`. Note that not all applications using currently know how to parse these protocols
`iso proto <protocol>`	Packet is an OSI packet of protocol `<protocol>`. Protocol can be a number or one of the names clnp, esis, or isis. Abbreviations for IS-IS PDU types are: l1, l2, iih, lsp, snp, csnp, psnp

The protocol can be a number or, e.g., one of the names `icmp`, `icmp6`, `igmp`, `igrp`, `pim`, `ah`, `esp`, `vrrp`, `udp`, or `tcp`. Note that the identifiers `tcp`, `udp`, and `icmp` are also keywords and must be escaped via backslash (), which is \ in the C-shell. Note that this primitive does not chase the protocol header chain.

`ip6 protochain 6` matches any IPv6 packet with TCP protocol header in the protocol header chain. The packet may contain, for example, an authentication header, routing header, or hop-by-hop option header between IPv6 header and TCP header. The BPF code emitted by this primitive is complex and cannot be optimized by the BPF optimizer code, which can be somewhat slow.

Ethernet protocol can be a number or one of the names ip, ip6, arp, rarp, atalk, aarp, decnet, sca, lat, mopdl, moprc, iso, stp, ipx, or netbeui. Note these identifiers are also keywords and must be escaped via backslash ().

In the case of FDDI (e.g., `fddi protocol arp`), Token Ring (e.g., `tr protocol arp`), and IEEE 802.11 wireless LANs (e.g., `wlan protocol arp`), for most of those protocols, the protocol identification comes from the 802.2 Logical Link Control (LLC) header, which is usually layered on top of the FDDI, Token Ring, or 802.11

headers.

When filtering for most protocol identifiers on FDDI, Token Ring, or 802.11, the filter checks only the protocol ID field of an LLC header in so-called SNAP format with an Organizational Unit Identifier (OUI) of 0x000000, for encapsulated Ethernet; it doesn't check whether the packet is in SNAP format with an OUI of 0x000000. The exceptions are:

- `iso` - the filter checks the DSAP (Destination Service Access Point) and SSAP (Source Service Access Point) fields of the LLC header
- `stp` and `netbeui` - the filter checks the DSAP of the LLC header
- `talk` - the filter checks for a SNAP-format packet with an OUI of 0x080007 and the AppleTalk etype

In the case of Ethernet, the filter checks the Ethernet type field for most of those protocols. The exceptions are:

- iso, stp, and netbeui - the filter checks for an 802.3 frame and then checks the LLC header as it does for FDDI, Token Ring, and 802.11
- atalk - the filter checks both for the AppleTalk etype in an Ethernet frame and for a SNAP-format packet as it does for FDDI, Token Ring, and 802.11
- aarp - the filter checks for the AppleTalk ARP etype in either an Ethernet frame or an 802.2 SNAP frame with an OUI of 0x000000
- ipx - the filter checks for the IPX etype in an Ethernet frame, the IPX DSAP in the LLC header, the 802.3-with-no-LLC-header encapsulation of IPX, and the IPX etype in a SNAP frame

1.1.7. Broadcast

Syntax	Condition
`ether broadcast`	Packet is an Ethernet broadcast packet. The ether keyword is optional
`ip broadcast`	Packet is an IPv4 broadcast packet
`ether multicast`	Packet is an Ethernet multicast packet. The ether keyword is optional. This is shorthand for `ether[0] & 1 != 0`
`ip multicast`	Packet is an IPv4 multicast packet
`ip6 multicast`	Packet is an IPv6 multicast packet

`ip broadcast` checks for both the all-zeros and all-ones broadcast conventions and looks up the subnet mask on the interface on which the capture is being done.
If the subnet mask of the interface on which the capture is being done is not available, either because the interface on which capture is being done has no netmask or because the capture is being done on the Linux *any* interface, which can capture on more than one interface, this check will not work correctly.

1.1.8. WLAN

Syntax	Condition
`wlan addr1 <ehost>`	First IEEE 802.11 address is `<ehost>`

Syntax	Condition
`wlan addr2 <ehost>`	Second IEEE 802.11 address, if present, is `<ehost>`. The second address field is used in all frames except for CTS (Clear To Send) and ACK (Acknowledgment) control frames
`wlan addr3 <ehost>`	Third IEEE 802.11 address, if present, is `<ehost>`. The third address field is used in management and data frames, but not in control frames
`wlan addr4 <ehost>`	Fourth IEEE 802.11 address, if present, is `<ehost>`. The fourth address field is only used for WDS (Wireless Distribution System) frames
`dir <dir`	IEEE 802.11 frame direction matches the specified dir. Valid directions are *nods*, *tods*, *fromds*, *dstods*, or a *snumeric* value
`type <wlan_type>`	IEEE 802.11 frame type matches the specified `<wlan_type>`. Valid WLAN types are *mgt*, *ctl* and *data*
`subtype <wlan_subtype>`	IEEE 802.11 frame subtype matches the specified `<wlan_subtype>` and frame has the type to which the specified WLAN subtype belongs
`type <wlan_type> subtype <wlan_subtype>`	IEEE 802.11 frame type matches the specified `<wlan_type>`and frame subtype matches the specified `<wlan_subtype>`. If the specified wlan_type is `mgt`, then valid wlan_subtypes are: *assoc-req*, *assoc-resp*, *reassoc-req*, *reassoc-resp*, *probe-req*, *probe-resp*, *beacon*, *atim*, *disassoc*, *auth*, and *deauth*. If the specified wlan_type is `ctl`, then valid wlan_subtypes are: *ps-poll*, *rts*, *cts*, *ack*, *cf-end*, and *cf-end-ack*. If the specified wlan_type is `data`, then valid wlan_subtypes are *data*, *data-cf-ack*, *data-cf-poll*, *data-cf-ack-poll*, *null*, *cf-ack*, *cf-poll*, *cf-ack-poll*, *qos-data*, *qos-data-cf-ack*, *qos-data-cf-poll*, *qos-data-cf-ack-poll*, *qos*, *qos-cf-poll* and *qos-cf-ack-poll*

1.1.9. VLAN

Virtual LAN (VLAN) IEEE 802.1Q tagged packets can be filtered with the `vlan` keyword. Filter `vlan <vlan_id>` yields packets that have the corresponding VLAN ID. Note that the first `vlan` keyword encountered in expression changes the decoding offsets for the remainder of the expression on the assumption that the packet is a VLAN packet. The `vlan <vlan_id>` expression may be used more than once to filter on VLAN hierarchies. Each use of that expression increments the filter offsets by 4.

For example, to filter on VLAN 200 encapsulated within VLAN 100:

```
vlan 100 && vlan 200
```

To filter IPv4 protocols encapsulated in VLAN 300 encapsulated within any higher order VLAN:

```
vlan && vlan 300 && ip
```

1.1.10. Multiprotocol Label Switching (MPLS)

Syntax	Condition
`mpls <label_num>`	Packet is an MPLS packet. If `<label_num>` is specified, the packet must have the corresponding label_num. Note that the first `mpls` keyword encountered in expression changes the decoding offsets for the remainder of the expression on the assumption that the packet is an MPLS-encapsulated IP packet. This expression may be used more than once to filter on MPLS hierarchies. Each use of that expression increments the filter offsets by 4

For example, filter packets with an outer label of 100000 and an inner label of 1024:

```
mpls 100000 && mpls 1024
```

Filter packets to or from 192.9.200.1 with an inner label of 1024 and any outer label:

```
mpls && mpls 1024 && host 192.9.200.1
```

1.1.11. Point-to-Point Protocol over Ethernet (PPPoE)

Syntax	Condition
`pppoed`	Packet is a PPP-over-Ethernet Discovery packet (Ethernet type 0x8863)
`pppoes`	Packet is a PPP-over-Ethernet Session packet (Ethernet type 0x8864). Note that the first pppoes keyword encountered in expression changes the decoding offsets for the remainder of expression on the assumption that the packet is a PPPoE session packet

For example, filter IPv4 protocols encapsulated in PPPoE:

```
pppoes && ip
```

1.2. Relation Expression

```
expr relop <expr>
```

True if the relation holds, where `relop` is one of `>`, `<`, `>=`, `<=`, `=`, `!=`, and `expr` is an arithmetic expression composed of integer constants (expressed in standard C syntax), the normal binary operators `+`, `-`, `*`, `/`, `&`, `|`, `<<`, `>>`, a length operator, and special packet data accessors. Note that all comparisons are unsigned, so that, for example, `0x80000000` and `0xffffffff` are `> 0`. To access data inside the packet, use the following syntax:

```
<proto> [ <expr> : <size> ]
```

Proto can be is one of `ether`, `fddi`, `tr`, `wlan`, `ppp`, `slip`, `link`, `ip`, `arp`, `rarp`, `tcp`, `udp`, `icmp`, `ip6` or `radio`, and indicates the protocol layer for the index operation. (`ether`, `fddi`, `wlan`, `tr`, `ppp`, `slip`, and `link` all refer to the link layer. `radio` refers to the *radio header* added to some 802.11 captures.) Note that `tcp`, `udp`, and other upper-layer protocol types only apply to IPv4, not IPv6 (this will be fixed in the future). The byte offset, relative to the indicated protocol layer, is given by `expr`. Size is optional and indicates the number of

bytes in the field of interest; it can be either one, two, or four and defaults to one. The length operator, indicated by the keyword `len`, gives the length of the packet.

For example, `ether[0] & 1 != 0` catches all multicast traffic. The `&`-sign means bit-wise masking, so the above expression basically checks the first byte's last bit 1. The expression `ip[0] & 0xf != 5` catches all IPv4 packets with options. The expression `ip[6:2] & 0x1fff = 0` catches only unfragmented IPv4 datagrams and frag zero of fragmented IPv4 datagrams. This check is implicitly applied to the `tcp` and `udp` index operations. For instance, `tcp[0]` always means the first byte of the TCP header and never means the first byte of an intervening fragment.

Some offsets and field values may be expressed as names rather than as numeric values. The following protocol header field offsets are available: `icmptype` (ICMP type field), `icmpcode` (ICMP code field), and `tcpflags` (TCP flags field).

The following ICMP type field values are available: `icmp-echoreply`, `icmp-unreach`, `icmp-sourcequench`, `icmp-redirect`, `icmp-echo`, `icmp-routeradvert`, `icmp-routersolicit`, `icmp-timxceed`, `icmp-paramprob`, `icmp-tstamp`, `icmp-tstampreply`, `icmp-ireq`, `icmp-ire-qreply`, `icmp-maskreq`, `icmp-maskreply`.

The following TCP flags field values are available: `tcp-fin`, `tcp-syn`, `tcp-rst`, `tcp-push`, `tcp-ack`, `tcp-urg`.

1.3. Combining Primitives

Primitives may be combined using:

- A parenthesized group of primitives and operators (parentheses are special to the Shell and must be escaped)
- Negation (`!` or `not`)
- Concatenation (`&&` or `and`)
- Alternation (`||` or `or`)

Negation has the highest precedence. Alternation and concatenation have equal precedence and associate left to right. Note that explicit and tokens, not juxtaposition, are now required for concatenation.

If an identifier is given without a keyword, the most recent keyword is assumed. For example, `not host vs and ace` is short for `not host vs and host ace`, which should not be confused with `not ( host vs or ace )`.

2. Examples

2.1. Basic and Common Scenarios

Next, some elementary examples of the use of Pcap syntax are given:

To get all traffic seen in the interface, enter an empty filter.

To get all IPv4 based traffic is:

```
ip
```

To get all traffic that involves my host (called myhost):

```
host myhost
```

To select all IPv4 traffic between 192.168.1.1 and 192.168.1.7:

```
ip host 192.168.1.1 and host 192.168.1.7
```

To select all IPv4 traffic between 192.168.1.1 and any host except 192.168.1.7:

```
ip host 192.168.1.1 and not host 192.168.1.7
```

To select all TCP traffic, including port 80 between 192.168.1.1 and 192.168.1.7:

```
ip host 192.168.1.1 and host 192.168.1.7 and tcp and port 80
```

To select all UDP traffic with even source ports between 192.168.1.1 and 192.168.1.7 or 192.168.1.6

```
ip and udp and (host 192.168.1.1 and (host 192.168.1.7) or (host 192.168.1.6)) and
(udp[0:2] & 1 = 0)
```

2.2. PROFINET over Ethernet

When running the PROFINET protocol in the real-time mode, it runs directly over Ethernet without IP. Currently, there is no automatic filtering for Ethernet packets, so one has to perform the filtering manually.

Assume that there are two hosts, *01:02:03:04:05:06* and *07:08:09:0a:0b:0c*, which communicate with PROFINET. If there is no other traffic traveling between the hosts, traffic between these hosts can be measured with a simple filter:

```
ether host 01:02:03:04:05:06 and ether host 07:08:09:0a:0b:0c
```

This will include all Ethernet traffic between the hosts. Thus, if there are, e.g., IP streams above Ethernet, those will also be included. When limiting the focus purely on PROFINET, this is done by a filter:

```
ether proto 0x8892
```

The reason for this kind of syntax is that PROFINET's protocol ID, or *Type* in Ethernet, is 8892 in hex. Pcap does not support PROFINET protocol directly. The filter above can be enough if one is sure that PROFINET communications take place only between the selected hosts. However, if there are more PROFINET streams, from the hosts to other hosts, the hosts' Ethernet addresses need to be included:

```
(ether host 01:02:03:04:05:06 and ether host 07:08:09:0a:0b:0c) and (ether proto 0x8892)
```

When using PROFINET, Virtual LANs are often used to improve QoS. This needs to be taken into account in the filter level as well, as discussed earlier. Thus, the final filter of this example will be in the form:

```
(ether host 01:02:03:04:05:06 and ether host 07:08:09:0a:0b:0c) and
((ether proto 0x8892) or (vlan and ether proto 0x8892))
```

2.3. Odd and Even Port Numbers

Capturing all UDP packets with an even-numbered port becomes topical when measuring RTP streams without RTCP-messages. Typically, RTP uses even-numbered ports and RTCP odd-numbered ones.

We cannot use the port statement since it cannot be manipulated in the way we need. Instead, we need to dig the protocol's port fields. By masking the least significant bit, we can reach our goal. A filter `udp[0:2] & 1 = 0` will take the first two bytes bits into checking, i.e., the source port. It masks that with 1, so when this is 0, the port must be even-numbered. Thus, the above filter takes all packets whose source port has an even-numbered value. Similarly, a filter `udp[2:2] & 1 = 0` includes all packets whose destination port is even-numbered. Then again, the filter `udp[2:2] & 1 = 1` includes all packets whose destination port is odd-numbered.

3. Special Cases

3.1. Sometimes the Order of Primitives Matter

There are some things to notice as Pcap libraries perform packet filtering in a particular order. For example, if there is a *VLAN* tag in the Ethernet header of a packet, at least some versions of Pcap won't go inside the packet at all if not especially told to do so. Thus, the packet content cannot be reached by usual filtering. Instead, insert `vlan and …` in front of the manual packet filter, and the filter works again as expected.

In some special two-point measurement setups, there can be VLAN tag in one end of the measurement, while not in the other. In this case, you cannot set a simple filter, but instead, the case with and without VLAN must be dealt with separately (with `or` statement) in the filter. For example, if you wish to measure all IP traffic, the filter would become:

```
ip or (vlan and ip)
```

It is important to notice here that in Pcap (at least in most of the Pcap versions), the order of the `vlan` statement in the filter matters! More specifically, the non-VLAN tagged part of the filter must be informed before the VLAN-tagged part. Otherwise, you will see only the VLAN-tagged part regardless of what comes next in the filter. Thus, if we write our previous example filter in the format:

```
(vlan and ip) or ip
```

which is logically the same as above, it will not work.

A good article related to this topic is given here.

If the two measurement points have different VLAN-setup, i.e., the other has a VLAN tag the other does not, you can also use auto-filtering. Just set the filter for the Primary Probe end and then select suitable autofilter mode. Notice: the autofiltering modes are currently quite simple, so if you have a complex filter, this won't work.

3.2. Curious Filtering Cases

A peculiar filtering behavior, at least, in some versions of Pcap (involving libpcap and Npcap), has been detected. Consider that you have two identical devices that both have two network interfaces. One interface has a fixed IP address of 192.168.0.1. The other interface is used (now in this example) for communications, and they have IP addresses, per device, of 192.168.2.100 and 192.168.2.101. Thus, Qosium's original automatic end-to-end filtering would yield a filter of

`ip and (host 192.168.0.1 or host 192.168.2.100) and (host 192.168.0.1 or host 192.168.2.101)` .

As seen, it has some redundancy since the IP address 192.168.0.1 is there twice, but from a logical point of view, it should work. It should give all IPv4-based traffic between the hosts 192.168.2.100 and 192.168.2.101, but it does not! Instead, it lets in only directional flows as if the filter would have been of form:

`ip and src host 192.168.2.100 and dst host 192.168.2.101` .

Thus, in general, if the filter is of form:

```
ip and (host <IP 3> or host <IP 1>) and (host <IP 3> or host <IP 2>)
```

, where the goal is to measure all IPv4-based traffic between <IP 1> and <IP 2>, it does not work as assumed. The causing factor is somehow related to the extra IP address (<IP 3>) being common to both sides of the filter. Instead, if the same filter is written in different order:

```
ip and (host <IP 1> or host <IP 3>) and (host <IP 2> or host <IP 3>)
```

, it works despite the fact that logically it should be the same as previous. Therefore, the order of primitives matters here. Then, of course, a filter:

```
ip and (host <IP 1> or host <IP 3>) and (host <IP 2> or host <IP 4>)
```

, works normally, as there is no common factor between the two sides of the filter that is, presumably, causing the issue. There is also a worse problem. Consider a filter of form:

```
ip and (host <IP 1> or host <IP 2> or host <IP 4>) and (host <IP 1> or host <IP 3> or host <IP
4>)
```

, or generally,

```
ip and (host <IP n1> or host <IP m1> or host … or host <IP M> or host <IP n2>) and (host <IP
n1> or host <IP k1> or host … or host <IP K> or host <IP n2>)
```

.

That kind of filter structure halts Pcap completely. The behavior has been detected, at least with some versions of libpcap. Npcap (at least from version 1.80 forward) does not halt, but it does not show any traffic either, even though, logically, it should.

The exact reasons for these detected behaviors of Pcap filtering are yet unknown to us. They could be some obvious normal consequences of the internal behavior of Pcap filter composition that we have missed or not.

Nevertheless, it shall be pointed out that none of the presented curious filtering cases are typically problematic: they are redundant, make no sense as such, and can be easily formed alternatively. When performing manual filtering, one will likely never face these issues. However, attention should be paid to automatic filtering. From Qosium Probe 1.9.2.0 onwards, these issues have been taken into consideration in the formation of the automatic filter. But, if you are building your own automatic monitoring setups with custom automatic parameterization, you also should pay attention to this.

4. Glossary

Virtual LAN

A LAN seqment which has no physical hardware, but instead is carried over network.