Per connection classifier

PCC matcher will allow you to divide traffic into equal streams with the ability to keep packets with specific set of options in one particular stream (you can specify this set of options from src-address, src-port, dst-address, dst-port)

Theory

PCC takes selected fields from IP header, and with the help of a hashing algorithm converts selected fields into 32-bit value. This value then is divided by a specified *Denominator* and the remainder then is compared to a specified *Remainder*, if equal then the packet will be captured. You can choose from src-address, dst-address, src-port, dst-port from the header to use in this operation.

PCC is not a valid method, in case of Hotspot(captive portal) - due to the fact that Hotspot uses web-proxy and it uses only the default routing table, at the moment.

```
per-connection-classifier=
PerConnectionClassifier ::= [!]ValuesToHash:Denominator/Remainder
  Remainder ::= 0..4294967295    (integer number)
  Denominator ::= 1..4294967295    (integer number)
  ValuesToHash ::= both-addresses|both-ports|dst-address-and-port|
  src-address|src-port|both-addresses-and-ports|dst-address|dst-port|src-address-and-port
```

Example

This configuration will divide all connections into 3 groups based on the source address and port

```
/ip firewall mangle add chain=prerouting action=mark-connection \
 new-connection-mark=1st_conn per-connection-classifier=src-address-and-port:3/0
/ip firewall mangle add chain=prerouting action=mark-connection \
  new-connection-mark=2nd_conn per-connection-classifier=src-address-and-port:3/1
/ip firewall mangle add chain=prerouting action=mark-connection \
  new-connection-mark=3rd_conn per-connection-classifier=src-address-and-port:3/2
```

How PCC works

This article aims to explain in simple terms how PCC works. The definition from the official manual wiki page reads: "PCC takes selected fields from IP header, and with the help of a hashing algorithm converts selected fields into 32-bit value. This value then is divided by a specified Denominator and the remainder then is compared to a specified Remainder, if equal then packet will be captured. You can choose from src-address, dst-address, src-port, dst-port from the header to use in this operation.", with the full number of fields available being: "both-addresses|both-ports|dst-address-and-port|src-address|src-port|both-addresses-and-ports|dst-address|dst-port|src-address-and-port". If you understand that definition, there'll be nothing interesting in this article for you.

First, here are the terms necessary to understand the definition.

IP packets have a header that contains several fields, two of those fields are the IP address of the source of the packet and the IP address of the destination of the packet. TCP and UDP packets also have headers that contain the source port and the destination port.

Denominators and remainders are parts of modulus operations. A modulus operation produces the integer left over when you divide two numbers and only accept the whole number portion of the result. It is represented by a % sign. Here are some examples: 3 % 3 = 0, because 3 divides cleanly by 3. 4 % 3 = 1, because the next smallest number to 4 that cleanly divides by 3 is 3, and 4 - 3 = 1. 5 % 3 is 2, because the next smallest number to 5 that divides cleanly by 3 is 3, and 5 - 3 = 2. 6 % 3 = 0, because 6 divides cleanly by 3.

A hash is a function that is fed input, and produces output. Hashes have many interesting properties, but the only important one for the purpose of this article is that hash functions are deterministic. That means that when you feed a hash function an input that reads 'hello' and it produces the output '1', you can rely on the fact that if you feed it 'hello' a second time it will produce the output '1' again. When you feed a hash function the same input, it will always produce the same output. What exact hashing algorithm is used by PCC is not important, so for this discussion let's assume that when you feed it IP addresses and ports, it just adds up the octets of the IP addresses as decimal numbers as well as the ports, and then takes the last digit and produces it as the output. Here an example:

The hash function is fed 1.1.1.1 as the source IP address, 10000 as the source TCP port, 2.2.2.2 as the destination IP address and 80 as the destination TCP port. The output will be 1+1+1+1+10000+2+2+2+2+80 = 10092, the last digit of that is 2, so the hash output is 2. It will produce 2 every time it is fed that combination of IP addresses and ports.

At this point it's important to note that even though PCC is most often used for spreading load across circuits, PCC itself has absolutely nothing to do with routing, routing marks or spreading load. PCC is simply a way to match packets, and not directly related to the action of then marking those matched packets even if that is its main purpose.

Here are three lines often used for PCC, with their explanation:

```
/ip firewall mangle add chain=prerouting action=mark-connection \
 new-connection-mark=1st_conn per-connection-classifier=src-address-and-port:3/0
/ip firewall mangle add chain=prerouting action=mark-connection \
  new-connection-mark=2nd_conn per-connection-classifier=src-address-and-port:3/1
/ip firewall mangle add chain=prerouting action=mark-connection \
  new-connection-mark=3rd_conn per-connection-classifier=src-address-and-port:3/2
```

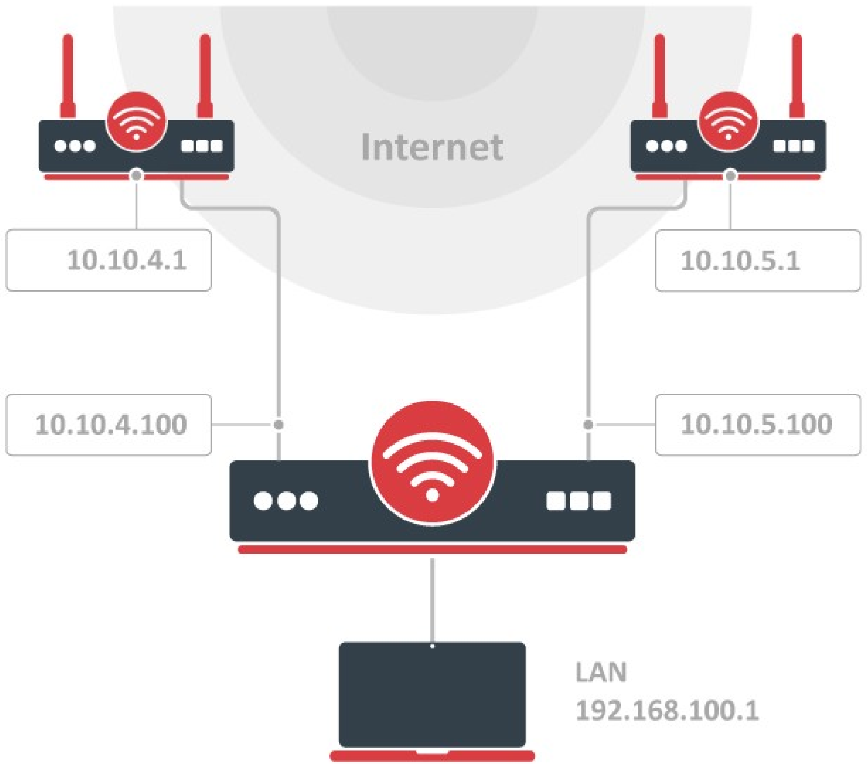
Here are what the different field options mean for the purpose of packet matching, these are the fields that will be fed into the hashing algorithm (and, for the purpose of spreading load across links, decide what link a packet will be put on). Remember that a hash function will always produce the same input when it's fed the same output:

- src-address: The source address of a client will always be the same, so all traffic from a particular client will always match the same PCC matcher, and will always be put on the same link.
- dst-address: The destination address of a specific server will always be the same, so all traffic to that server (say, the Mikrotik Wiki) will always match the same PCC matcher, and will always be put on the same link.
- both-addresses: The source and destination IP pair between the same client and server will always be the same, so all traffic between one specific client and a specific server (say, your laptop and the Mikrotik Wiki) will always match the same PCC matcher, and will always be put on the same link.
- src-port: Source ports of clients are usually randomly chosen when the connection is created, so across many connections different source ports will be fed into the hash function, and different PCC matchers will match and traffic will go across different links. However, some client protocols always choose the same source port, and servers behind your router will mostly likely always use the same service port to send traffic back to their clients. A web server behind your router would send most traffic from its HTTP (80) and HTTPS (443) ports, and that traffic would always match the same PCC matcher and would be put on the same link.
- dst-port: Destination ports of clients are usually well defined service ports, all HTTP (80) traffic between your clients and servers on the Internet would always match the same PCC matcher, and would be put on the same link. However, the same clients doing HTTPS (443) traffic could match a different PCC matcher, and would go across a different link.
- both-ports: Since the client port is (usually) randomly chosen, the combination of the two ports is (usually) random and will spread load across links.
- src-address-and-port: Same caveat as src-port.
- dst-address-and-port: Same caveat as dst-port.
- both-addresses-and-ports: This is the most random way to spread traffic across links, since it has the most number of variables.

It's important to note that even though the hash function discussed in this article is greatly simplified and not what is used in real life, it nicely demonstrates another property of hash functions: two completely different inputs can produce the same output. In our example, 3 % 3 = 0, and 6 % 3 = 0; we get back 0 when we feed it a 3 as well as when we feed it a 6. The same is true for the actual function used for PCC, even though I don't know what it is we do know from the definition that it produces a 32 bit value as output. IP addresses are 32 bit, and ports are 16 bit, so assuming that we're using both-addresses-and-ports, we'd be feeding it 32+32+16+16 = 96 bits of input and would only receive 32 bits back, so it must be producing the same output for different inputs. This means that two completely unrelated connections could match the same PCC matcher, and would be put on the same line. PCC works better the more connections you put across it so that the hash function has more chances to produce different outputs.

Configuration Example

Let's assume this configuration:



IP Addresses

```
/ip address
add address=10.10.4.100/24 interface=ether_ISP1 network=10.10.4.0
add address=10.10.5.100/24 interface=ether_ISP2 network=10.10.5.0
add address=192.168.100.1/24 interface=ether_LAN network=192.168.100.0
```

The router has two upstream (ISP) interfaces with the addresses of 10.10.4.100/24 and 10.10.5.100/24. The LAN interface has IP address of 192.168.0.1 /24.

We are adding two new Routing tables, which will be used later:

```
/routing table
add disabled=no fib name=ISP1_table
add disabled=no fib name=ISP2_table
```

Policy routing

```
/ip firewall mangle
add action=accept chain=prerouting dst-address=10.10.4.0/24 in-interface=ether_LAN
add action=accept chain=prerouting dst-address=10.10.5.0/24 in-interface=ether_LAN
```

With policy routing it is possible to force all traffic to the specific gateway, even if traffic is destined to the host (other that gateway) from the connected networks. This way routing loop will be generated and communications with those hosts will be impossible. To avoid this situation we need to allow usage of default routing table for traffic to connected networks.

```
add action=mark-connection chain=input connection-state=new in-interface=ether_ISP1 new-connection-mark=ISP1
add action=mark-connection chain=input connection-state=new in-interface=ether_ISP2 new-connection-mark=ISP2

add action=mark-connection chain=output connection-mark=no-mark connection-state=new new-connection-mark=ISP1
passthrough=yes per-connection-classifier=both-addresses:2/0
add action=mark-connection chain=output connection-mark=no-mark connection-state=new new-connection-mark=ISP2
per-connection-classifier=both-addresses:2/1
```

First it is necessary to manage connection initiated from outside - replies must leave via same interface (from same Public IP) request came. We will mark all new incoming connections, to remember what was the interface.

```
add action=mark-connection chain=prerouting connection-mark=no-mark connection-state=new dst-address-type=!
local in-interface=ether_LAN new-connection-mark=ISP1 per-connection-classifier=both-addresses:2/0
add action=mark-connection chain=prerouting connection-mark=no-mark connection-state=new dst-address-type=!
local in-interface=ether_LAN new-connection-mark=ISP2 per-connection-classifier=both-addresses:2/1
```

Action mark-routing can be used only in mangle chain output and prerouting, but mangle chain prerouting is capturing all traffic that is going to the router itself. To avoid this we will use dst-address-type=!local. And with the help of the new PCC we will divide traffic into two groups based on source and destination addressees.

```
add action=mark-routing chain=output connection-mark=ISP1 new-routing-mark=ISP1_table
add action=mark-routing chain=prerouting connection-mark=ISP1 in-interface=ether_LAN new-routing-mark=ISP1_table

add action=mark-routing chain=output connection-mark=ISP2 new-routing-mark=ISP2_table
add action=mark-routing chain=prerouting connection-mark=ISP2 in-interface=ether_LAN new-routing-mark=ISP2_table
```

Then we need to mark all packets from those connections with a proper mark. As policy routing is required only for traffic going to the Internet, do not forget to specify in-interface option.

```
/ip route
add check-gateway=ping disabled=no dst-address=0.0.0.0/0 gateway=10.10.4.1 routing-table=ISP1_table suppress-hw-
offload=no
add check-gateway=ping disabled=no dst-address=0.0.0.0/0 gateway=10.10.5.1 routing-table=ISP2_table suppress-hw-
offload=no
```

Create a route for each routing-mark

```
add distance=1 dst-address=0.0.0.0/0 gateway=10.10.4.1
add distance=2 dst-address=0.0.0.0/0 gateway=10.10.5.1
```

To enable failover, it is necessary to have routes that will jump in as soon as others will become inactive on gateway failure. (and that will happen only if check-gateway option is active)

NAT

```
/ip firewall nat
add action=masquerade chain=srcnat out-interface=ether_ISP1
add action=masquerade chain=srcnat out-interface=ether_ISP2
```

As routing decision is already made we just need rules that will fix src-addresses for all outgoing packets. If this packet will leave via ether_ISP1 it will be NATed to 10.10.4.100, if via ether_ISP2 then NATed to 10.10.5.100