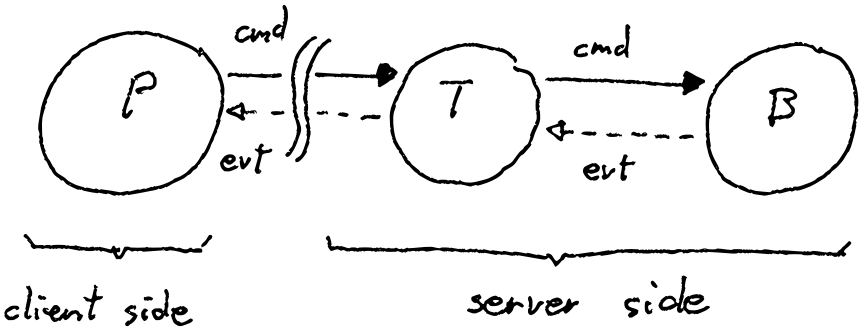
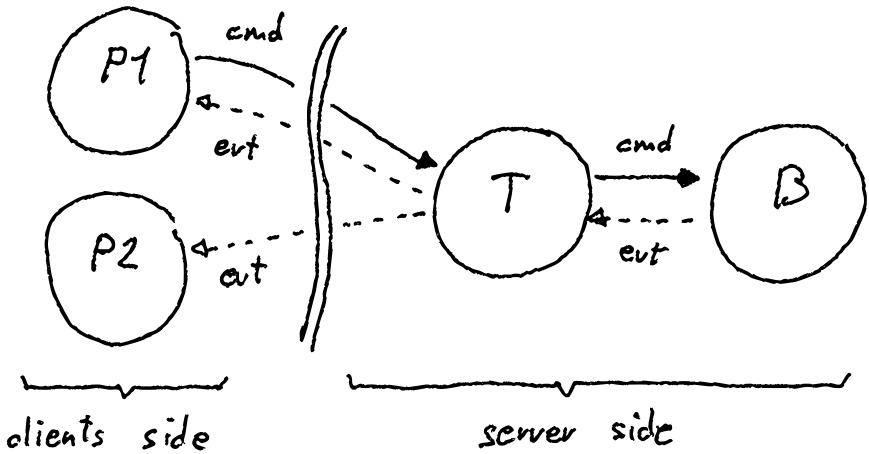
# Multicasting with Xcraft

By multicasting we talk about the ways where the events must follow from a sender to one (unicast) or multiple (multicast) receivers. The original implementation was using only the broadcast. For performances and security reasons, the multicast was necessary.
In order to achieve this goal, we have implemented a complexe mechanic in the router stuff (see xcraft-core-transport).
To multicast we must know which are the clients and where they are. For that it uses table "ARP" where a socket is mapped with a client (orcName). Then we know that by sending the message to this socket, it should be possible to reach the client. But in order to understand this stuff, a diagram will be (for sure) very useful.




Here, P is the client. He sends a command to the server which is exposed on Internet via T. T doesn't know this command then it sends this one to B.
When B sends the result, the event is forwarded to T, and T forwards to P.
This example is very simple. Here the same idea but with two clients.

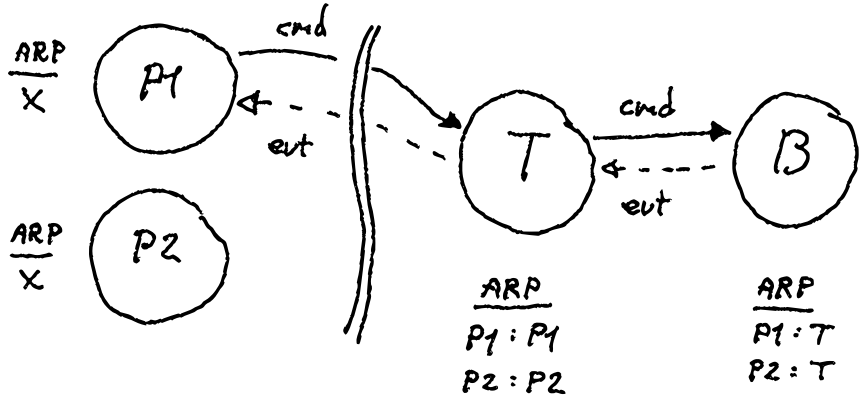


With this diagram, the multicast is still not implemented. It's not like the first diagram where it is impossible to tell if it's uni, multi or broadcast.

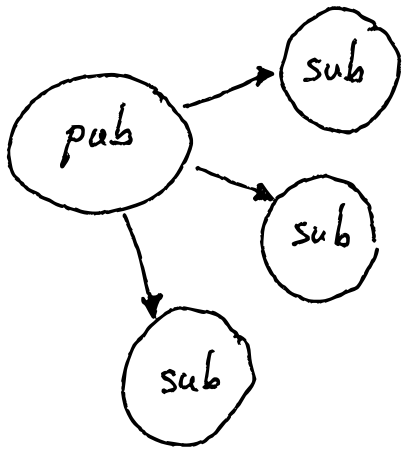
For this second diagram, the event is sent only one time by B, but T knows more than one client then it sends (duplicate) for everybody.

Here the problems begin. It's not safe, it's "slow" (it depends of how many clients are connected) but it's simple. No need to know the clients.

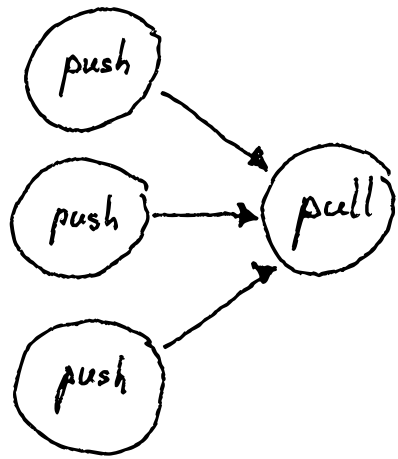
It's not acceptable then the multicast is necessary and must be implemented a bit like for an usual network.



Each node has an ARP table. In these tables the clients are mapped with a socket.
The tables of P1 and P2 are empty because nobody is connected on these nodes.
T knows two clients (P1 and P2) with a direct connection (it's directly the socket of P1 and P2). But in the case of B, P1 and P2 are mapped on the same node which is T.
Fine, but how are the tables populated ?
It uses the commands... because the commands use an other topology. Events are pub/sub and the commands are push/pull.




1 sender, n receivers

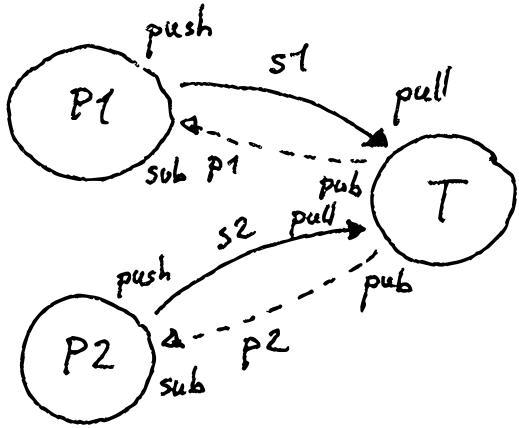



n senders, 1 receiver

When a command is received by a node, the router can update it's ARP table with some informations. The table has two types of routers, then a client can be in the Event Emitter type, in the Axon type or in both. For each type we can see the same structure. The key is the client id (orcName) and the value is the server token (direct destination for this client), the socket (net.Socket for Axon or an id for EventEmitter), the port where we can send the events (this port is used in order to retrieve the appropiate socket), the list of hordes which are known by the server behind the token/socket, the nice value for the priorities and the multicast lines for the switching. The socket is only used for receiving the commands.
About the nice value, it's avalaible in the tables and it's propagated where appropriate but it's not used (WIP).

# ARP Table in T

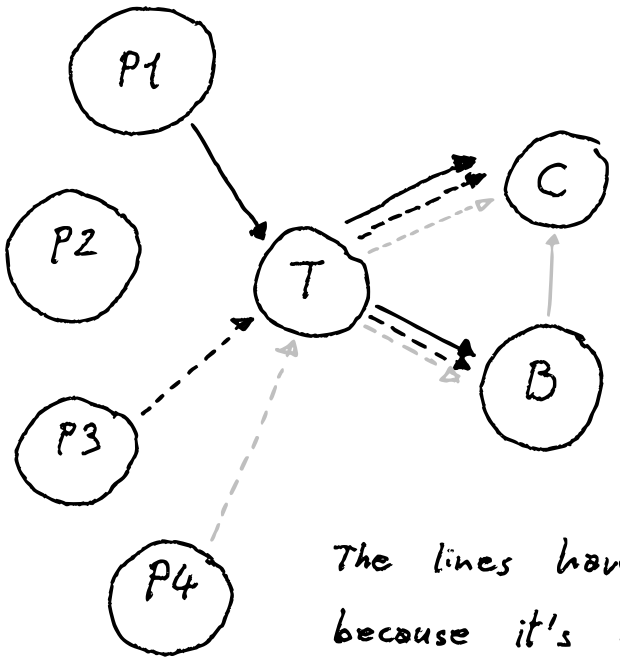
| Bakend | orcName | token | sock | port | horde | nice | lines |
|---|---|---|---|---|---|---|---|
| Axon | orcP1 | P1 | s1 | p1 | T | -4 | — |
| | orcP2 | P2 | s2 | p2 | T | -4 | — |




P1 push cmd via s1 to T
P2 push cmd via s2 to T
T pull cmds from s1, s2
P1 sub evt via p1 on T
P2 sub evt via p2 on T
T pub evts to p1, p2

With the ARP Table, it's easy to have the relationship between a command socket and an event port for a specific client (orcName). Now that you have a better idea about the topologies and the links between the nodes, we can study the lines which are used by the multicasting.

A line is an ID used to create relationships between the nodes in order to send events with the "most" efficient way. Without lines, the events are sent in broadcast or in unicast for some very specific events (which would be translated to lines in the future...).




P1 —> B, C

P2

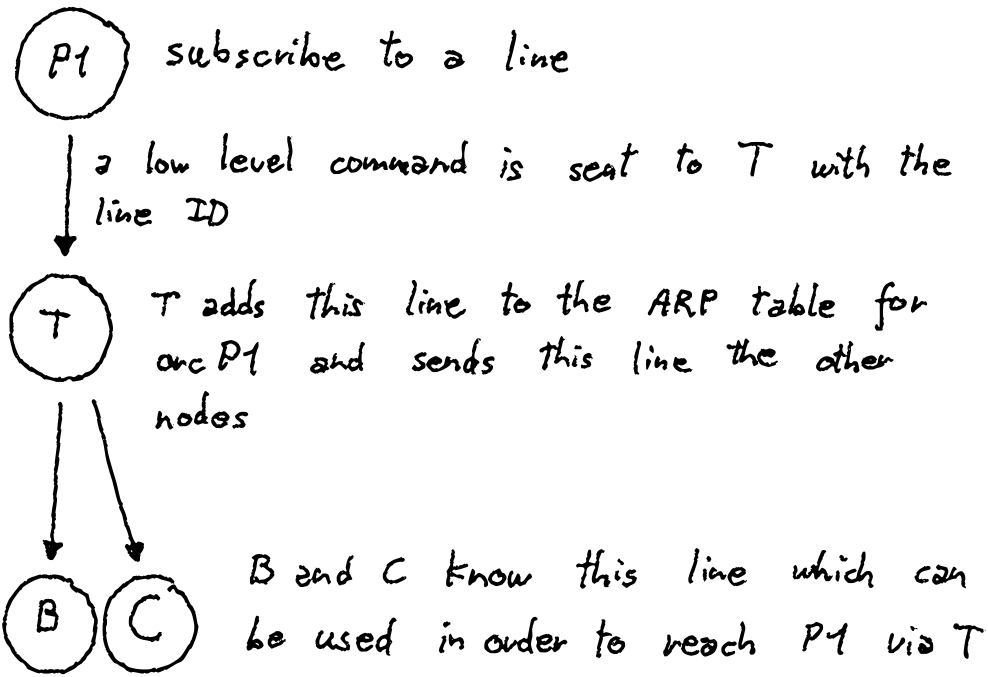
P3 —> B, C

P4 —> B, C

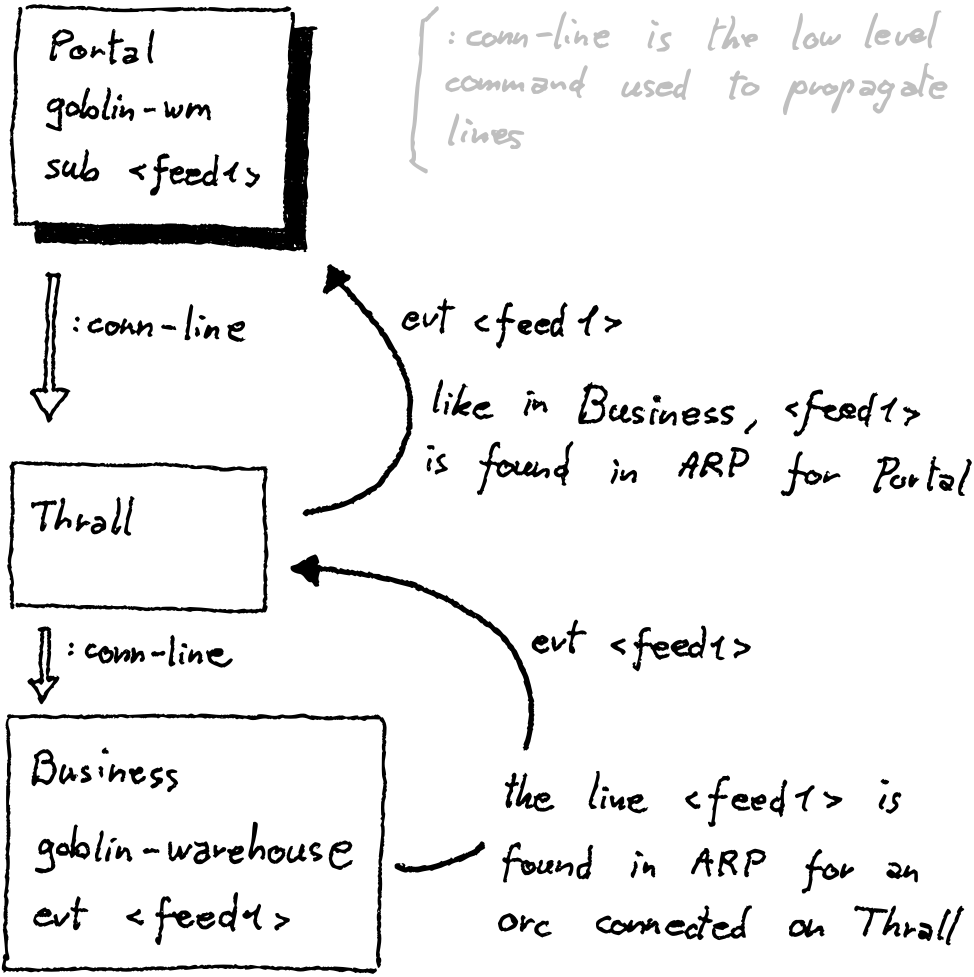
B —> C

The lines have a direction because it's used in order to know where is a client for a specific ID (subscription). When P1 sends its line to B, it tells to T that P1 is the destination for ID, and T tells to B that P1 can be reached via T for this ID, etc,...

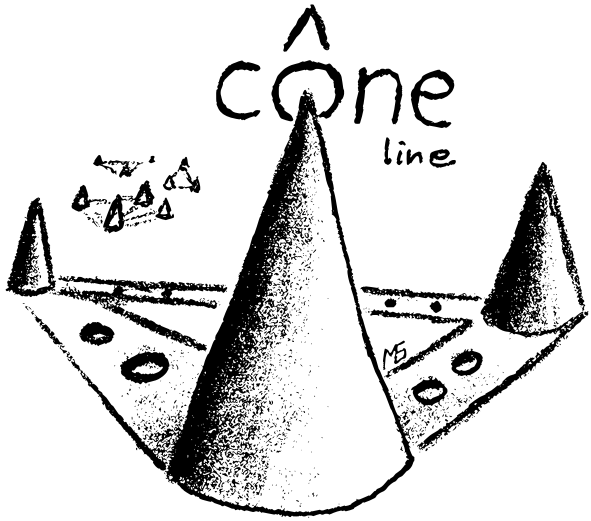
The lines can be used for special links like B and C for example. As you can see in the diagram, C knows B and can sends events via a line, and this line is not visible for P1, P2, P3 and P4
The next diagramm shows only one line with more details.



A more interesting example will be the changes sent by warehouse to the wm which has subscribed.




cryo is ignored here because even if it knows the line, it doesn't use it.


cône
line

As previously explained, the :conn-line internal command is sent when subscribing to events with a line Id. Of course, when the subscriber calls the unsubscribe an other command is sent by the way ":disconn-line". An other possibility exists. Instead of an unsubscribe, the whole client can be removed. In this case it sends an other internal command that is used in order to remove a route ":delete-route". When a route entry is deleted, the whole oneName is removed from the ARP table (the lines are removed by the way).

Delete Route