

Figure 1 shows a small fully meshed mcrover team. It is drawn as a complete digraph because alerts only travel in one direction for each TCP connection; there are two TCP connections between each pair of mcrover team members, one for each direction of alert travel.

Note that mcrover team members do not receive forwarded alerts. A given TCP connection between two team members will only carry alerts that were created on the directly connected team host. In Figure 1, this means that the blue line from Host 1 to Host 3 will only transport alerts originated at Host 1. Alerts sent from Host 2 to Host 1 will not be seen by Host 3. However, that's why we have a full mesh in this configuration. Host 3 will receive alerts from Host 2 via a direct connection from Host 2.

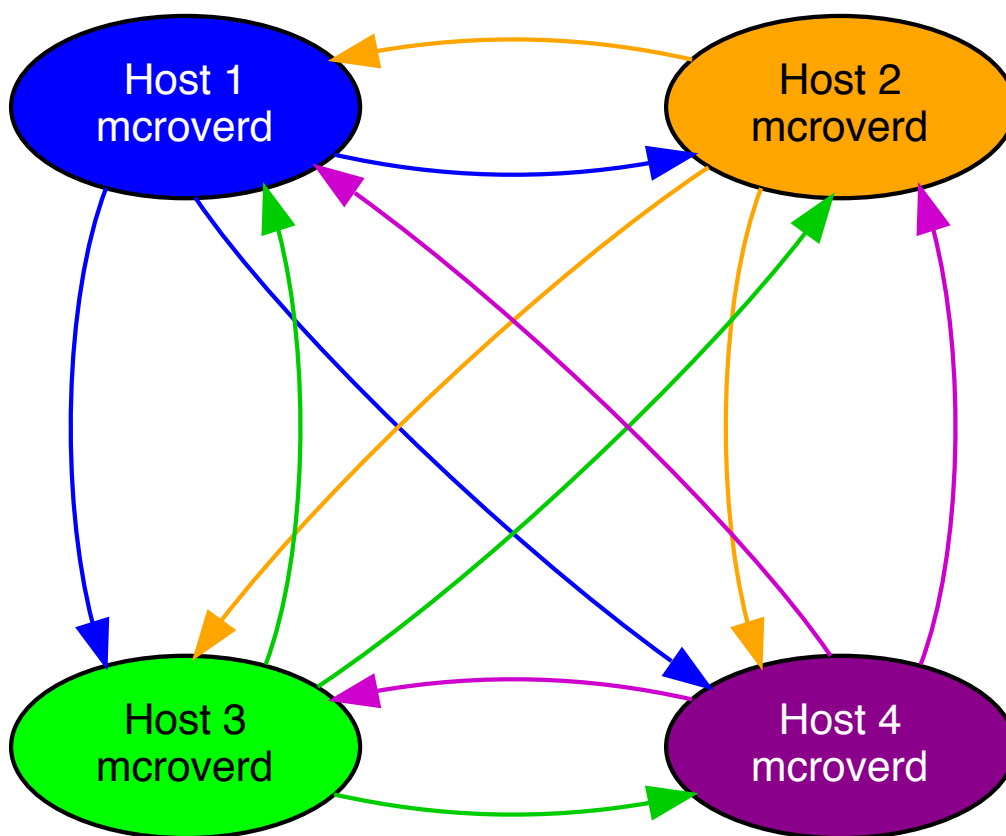


Figure 1: mcroverd full mesh

A mcrover display client, such as `mcrover` or `qmcrover`, will receive all alerts from its peer. This allows a display

client to see all alerts via a single connection to a mcrover team member. See **Figure 2**.

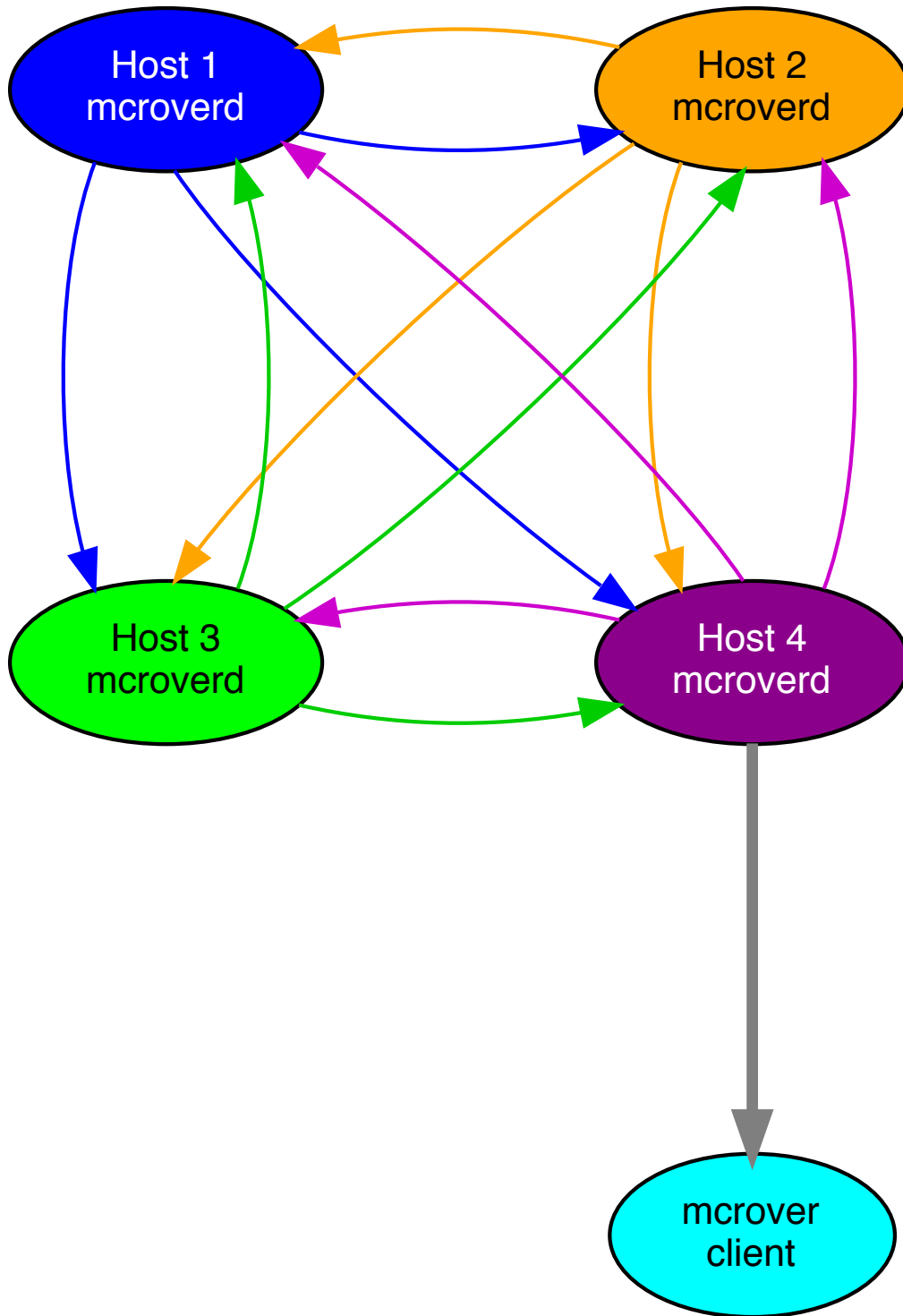


Figure 2: mcoverd full mesh with display client

It would be nice if I could do something like [Figure 3](#). This requires more smarts, however. I'd need more logic for alert forwarding. And before someone asks... I don't want to use BGP.

With this particular topology, a single mcroverd instance within a team will forward all of its team's alerts to other teams. It will not forward alerts from an external team to another external team.

An advantage here is just privacy for the use case I have in mind: connecting mcroverd instances in the home of friends and family. You only share alerts with those you trust, and they don't forward them outside of their team. The logic is also not tricky here, since we only need to distinguish between alerts from two types of peers. No policy configuration needed.

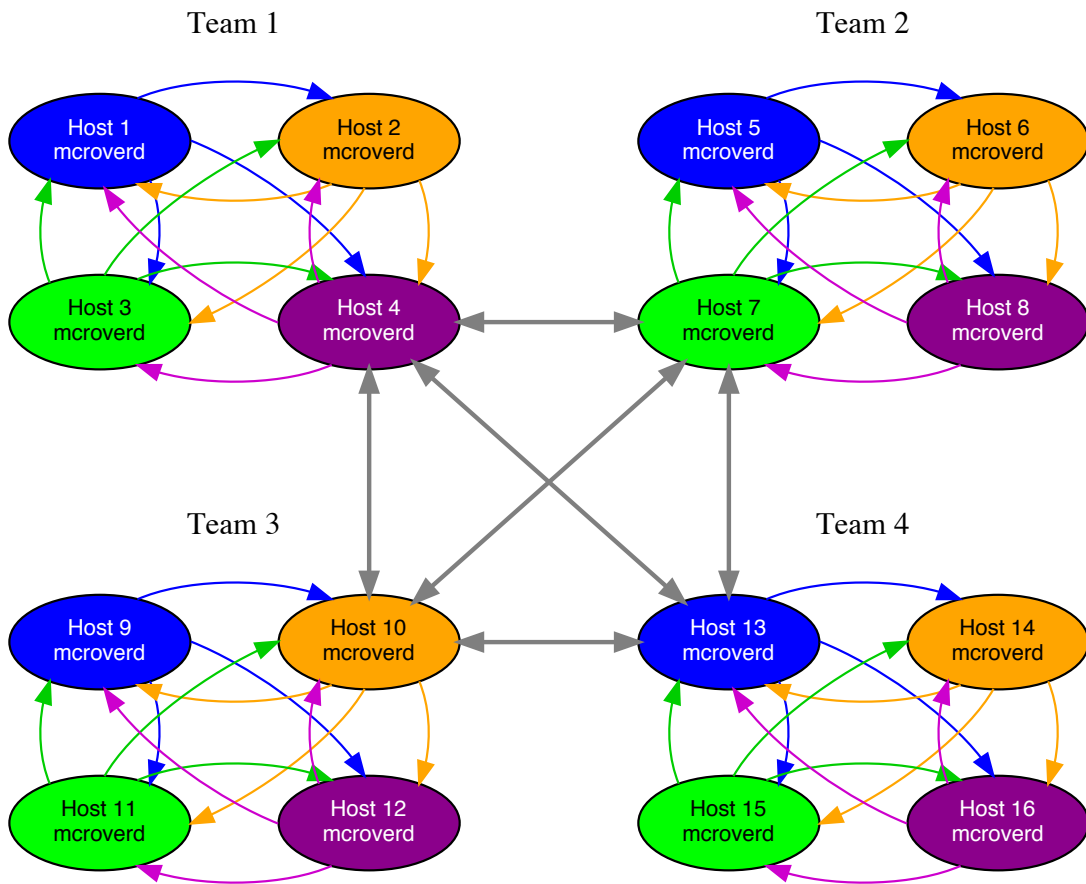


Figure 3: mcroverd federated mesh

[Figure 4](#) shows a partially connected setup. This is a case that doesn't let any mcroverd see all alerts, unless I implement more logic. Here, Team 3 will not see alerts from Team 2 or Team 4. I honestly think that's as it should

be, but it means that as-is, we can't scale to huge deployments.

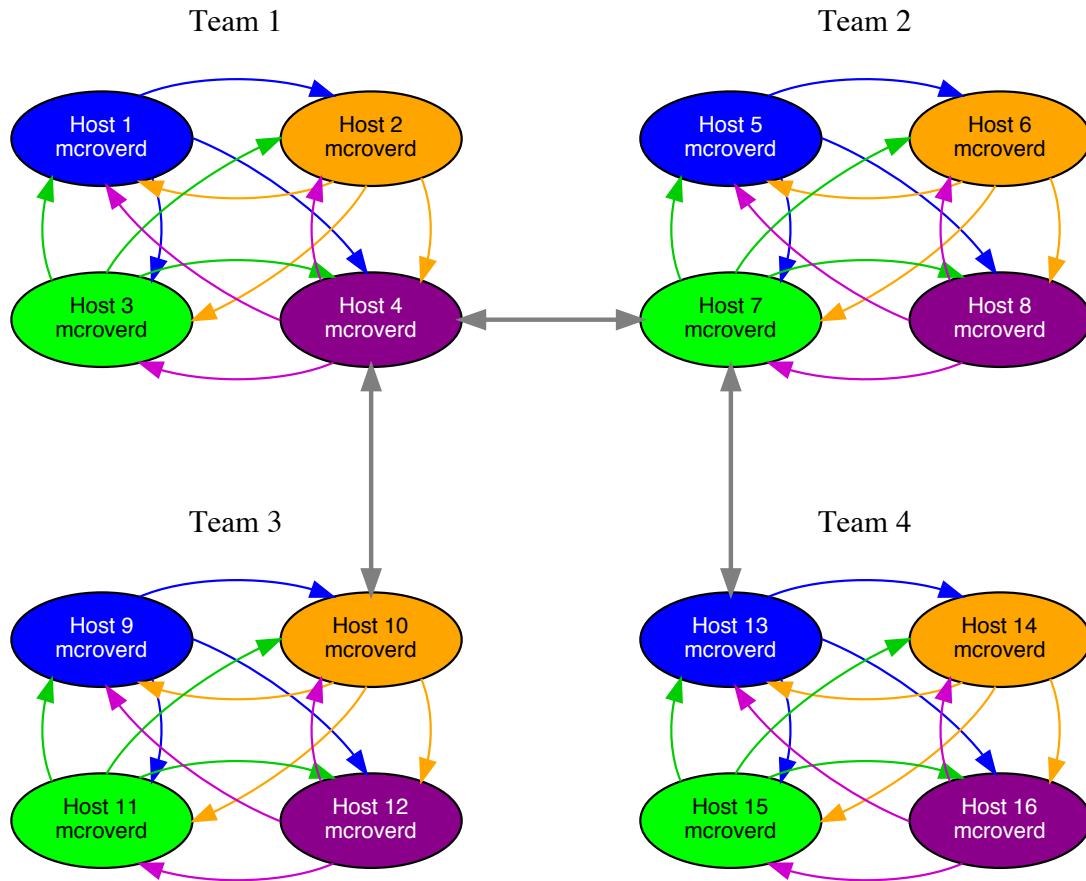


Figure 4: mcoverd not meshed

I'm OK with the constraint that we only see alerts from our own team and directly connected teams. So if I run with this, the logic becomes:

Source	Destination(s)
Self	My team, other teams
My team	Other teams
Other teams	My team

It's worth noting that I'd expect mcoverd hosts to be NATed. This means that IP addresses may conflict between teams. Hence we must maintain a (team,host) tuple for each alert.

This allows a fairly simple (and shallow) alert topology.

```
AllAlerts
  MyTeam_Alerts
    My_Alerts
    MyTeam_Member1_Alerts
    ...
    MyTeam_MemberN_Alerts
  TeamB_Alerts
    TeamB_Member1_Alerts
    ...
    TeamB_Member2_Alerts
  TeamC_Alerts
    TeamC_Member1_Alerts
    ...
    TeamC_Member2_Alerts
  ...
  TeamZ_Alerts
```

I need a small set of commands to be able to communicate with my peers. UpdateAlert, RemoveAlert, RemoveTeamAlerts.

UpdateAlert would update a single alert, possibly adding it if it doesn't already exist at the receiver.

RemoveAlert would remove a single alert.

RemoveTeamAlerts would remove all alerts from a given team. The only time this should be used would be when we lose our connection to another team; we would remove all alerts we've seen from that team (for ourselves and all of our team members), and add a ROVER alert for the other team, sourced from us.

There's an issue here. If we're just a team member (no direct connections to another team), how do we easily remove external alerts we received from a team member when we lose our connection to that team member?

What if we use an alert path? If we look back at [Figure 3](#), an alert from Host 6 to Host 11 would have an alert path of *Host6* → *Host7* → *Host10* → *Host11*. If the connection from Host 10 to Host 11 is lost, Host 11 should remove all alerts which have Host 10 in the path.

This means we need a different alert heirarchy.

While we're here, what could we do to handle the topology of [Figure 4](#) and still have all alerts visible, while avoiding loops in the topology of [Figure 3](#)? I don't think this is terribly difficult on the surface; if we already have an alert from a given origin, regardless of its path, we don't insert nor redistribute the alert to others; it's the same alert, it just arrived via a different path. Let's say Host 1 originates a ZFS alert. It sends it to its team members, which includes Host 4. Host 4 sends it to Host 7, Host 10 and Host 13. Host 10 then sends it to Host 7. Host 7 should ignore it because it already received the same alert from Host 4.

This turns the alert heirarchy upside down; our primary key should be the origin host. Code-wise this winds up looking a lot like a typical BGP RIB implementation; we have a top-level hash table keyed by the origin address, and each value in the hash table is the alert data plus a vector (or hash table) of the paths from which we received the alert.

Upsides to this approach...

We get full alert distribution.

Can we still remove all alerts with a given host in the path whenever we lose our direct connection to that path? I think so...

Downsides to this approach...

We still forward alerts to hosts that don't need them. This is a flooding strategy, and doesn't have optimal convergence nor optimal bandwidth utilization. Solving this one adds a decent chunk of logic. I really don't want to have to distribute something like OSPF LSAs and build a graph. On the other hand, if I know all of the possible paths to get an alert from one host to another, I could just pick one path and forward the alerts along that path. Remembering that I only need to get the alert to each team, I really only need to know the paths to each team leader (sort of akin to an OSPF area border router). I don't think I need metrics for each direct connection; they could all be 1? In any event, when a team border mcroverd sends an alert to another team's border mcroverd, it could include the list of other team members to which it is connected. The receiving border mcroverd would not forward the alert to any border mcroverd from the list. In [Figure 3](#), when mcroverd sends an alert to Host 7, it would tell Host 7 to not send the alert to Host 10 or Host 13 since Host 4 will send the alert directly to those hosts.

We would forward the list throughout.

We'd still have a problem with, say, a diamond border topology like [Figure 5](#). Alerts sent from Host 4 to Host 7 and Host 10 would still both be forwarded to Host 13, for example. This is just more evidence that for general optimal forwarding, we actually need to know the topology. When we see equal cost multipaths, we need to choose one and provide a means of forwarding only along that path.

Fortunately, we have things like the Boost Graph Library (BGL), so this isn't the end of the world. However, even if we distribute link states, we still need to control their distribution. The good news is that the full set of border-to-border mcroverd links would normally be small. Assuming the state doesn't change frequently, flooding with only a stop on "hey, I originated this list" is probably very workable. Flooding with a stop on "the peer is already in the path that has seen this set of link states" should be very workable.

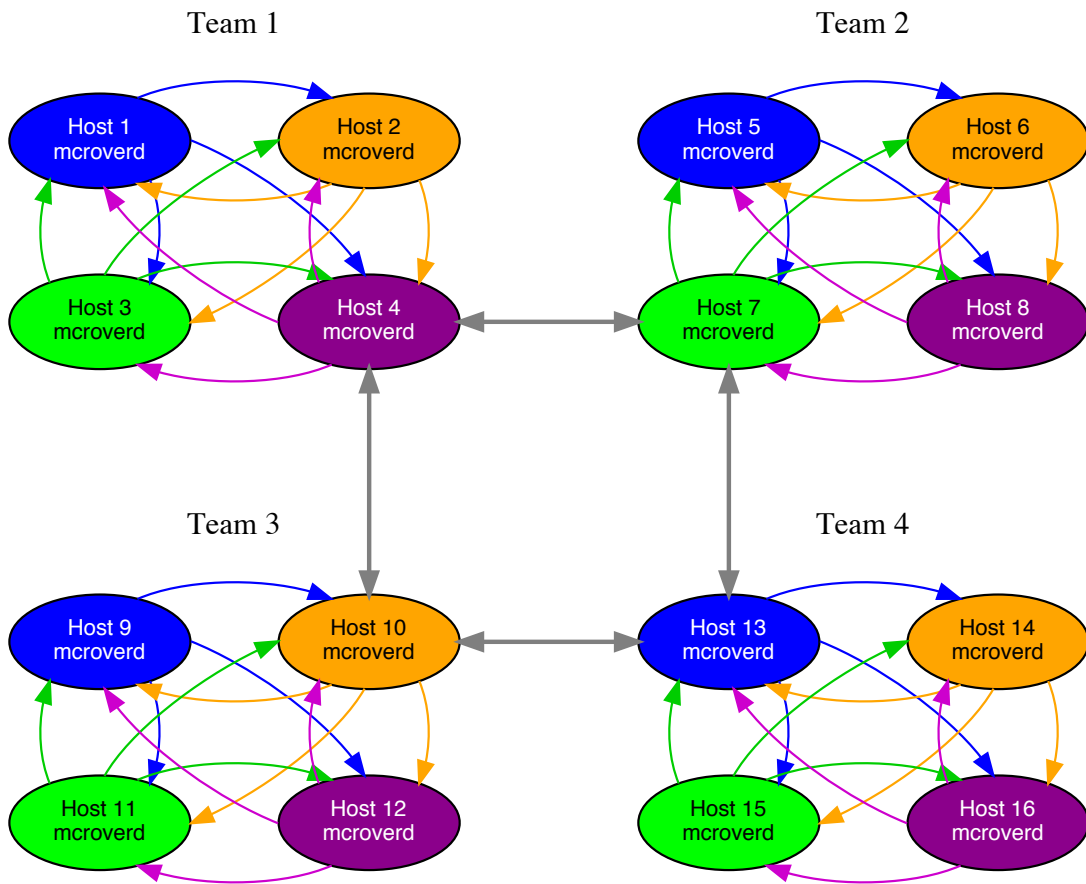


Figure 5: mcoverd border hosts in diamond