Test fault tolerance and Chat GPT security

Prompt

I have this network which is in csv format exported from neo4j, a network created with the cypher language. I want to make this network fault tolerant, give me the commands in cypher to change the network so that it is fault tolerant.

n,r,m
"(:Switch {grpc_address: 127.0.0.1:50051,device_id: 0,json_path: build/basic.json,model: bmv2,id: s1})","[:INTERFACE {port: 1,name: s1-eth1,mac: 00:11:22:33:44:55}]","(:Host {ip: 10.0.0.1/24,id: h1,default_gw: 10.0.0.254})"
"(:Switch {grpc_address: 127.0.0.1:50051,device_id: 0,json_path: build/basic.json,model: bmv2,id: s1})",[:INTERFACE {name: s1-eth2}],"(:Host {ip: 10.0.0.2/24,id: h2,default_gw: 10.0.0.254})"
"(:Host {ip: 10.0.0.1/24,id: h1,default_gw: 10.0.0.254})",[:LINKS_TO],"(:Switch {grpc_address: 127.0.0.1:50051,device_id: 0,json_path: build/basic.json,model: bmv2,id: s1})"
"(:Host {ip: 10.0.0.2/24,id: h2,default_gw: 10.0.0.254})",[:LINKS_TO],"(:Switch {grpc_address: 127.0.0.1:50051,device_id: 0,json_path: build/basic.json,model: bmv2,id: s1})"
"(:Switch {grpc_address: 127.0.0.1:50051,device_id: 0,json_path: build/basic.json,model: bmv2,id: s1})",[:LINKS_TO {via: uplink-s1-s2}],"(:Switch {grpc_address: 127.0.0.2:50051,device_id: 1,json_path: build/basic.json,model: bmv2,id: s2})"

Answer 1

Today your network has a single main switch (s1) connected to two hosts (h1, h2) and an uplink to s2. The problem is that if s1 goes down, the whole network goes down with it.

To become fault tolerant, we need to:

1. **Add switch redundancy (connect h1 and h2 to s2 as well).**
2. **Create additional links between s1 and s2 (to have alternative paths).**
3. **(Optional) Add a switch s3 and connect it to the mesh, but I can leave that for a second step if you like.**

```
// Create extra interfaces in s2 for the hosts
MATCH (s2:Switch {id: 's2'}), (h1:Host {id: 'h1'})
MERGE (s2)-[:INTERFACE {port: 1, name: 's2-eth1', mac: '00:11:22:33:44:66'}]->(h1)
MERGE (h1)-[:LINKS_TO]->(s2);

MATCH (s2:Switch {id: 's2'}), (h2:Host {id: 'h2'})
MERGE (s2)-[:INTERFACE {port: 2, name: 's2-eth2', mac: '00:11:22:33:44:77'}]->(h2)
MERGE (h2)-[:LINKS_TO]->(s2);

// Create second redundant uplink between s1 and s2
MATCH (s1:Switch {id: 's1'}), (s2:Switch {id: 's2'})
MERGE (s1)-[:LINKS_TO {via: 'uplink-s1-s2-backup'}]->(s2)
MERGE (s2)-[:LINKS_TO {via: 'uplink-s2-s1-backup'}]->(s1);

// Optional: add direct link between hosts (last resort for redundancy)
MATCH (h1:Host {id: 'h1'}), (h2:Host {id: 'h2'})
MERGE (h1)-[:LINKS_TO {via: 'direct-h1-h2'}]->(h2)
MERGE (h2)-[:LINKS_TO {via: 'direct-h2-h1'}]->(h1);
```

What changes

- h1 and h2 will have two paths to the network (s1 and s2).
- s1 and s2 will have two redundant uplinks.
- If s1 goes down, the hosts will still communicate via s2.
- The direct link h1 ↔ h2 is optional, just to give an extra path if the switches fail.

Now let's add a third switch (s3) and form a triangular topology between s1, s2 and s3. This way, even if a switch fails, there will always be an alternative path.

```
// Create the s3 switch
MERGE (s3:Switch {
  id: 's3',
  grpc_address: '127.0.0.3:50051',
  device_id: 2,
  json_path: 'build/basic.json',
  model: 'bmv2'
});

// Connect s3 to hosts for redundancy
MATCH (s3:Switch {id: 's3'}), (h1:Host {id: 'h1'})
MERGE (s3)-[:INTERFACE {port: 1, name: 's3-eth1', mac: '00:11:22:33:44:88'}]->(h1)
MERGE (h1)-[:LINKS_TO]->(s3);

MATCH (s3:Switch {id: 's3'}), (h2:Host {id: 'h2'})
MERGE (s3)-[:INTERFACE {port: 2, name: 's3-eth2', mac: '00:11:22:33:44:99'}]->(h2)
MERGE (h2)-[:LINKS_TO]->(s3);

// Create links between s3 and the other switches
MATCH (s1:Switch {id: 's1'}), (s3:Switch {id: 's3'})
MERGE (s1)-[:LINKS_TO {via: 'uplink-s1-s3'}]->(s3)
MERGE (s3)-[:LINKS_TO {via: 'uplink-s3-s1'}]->(s1);

MATCH (s2:Switch {id: 's2'}), (s3:Switch {id: 's3'})
MERGE (s2)-[:LINKS_TO {via: 'uplink-s2-s3'}]->(s3)
MERGE (s3)-[:LINKS_TO {via: 'uplink-s3-s2'}]->(s2);
```

Hosts (h1, h2) connected to three switches (s1, s2, s3).

Malha triangular (s1–s2–s3) → each switch has two alternative paths.

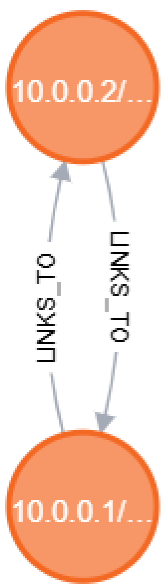
Fault tolerance:

- If a switch goes down, the hosts still reach the network.
- If an uplink goes down, there are other links available.

Cypher query for you to visualize all redundant paths (e.g. all possible paths between h1 and h2), to confirm fault tolerance

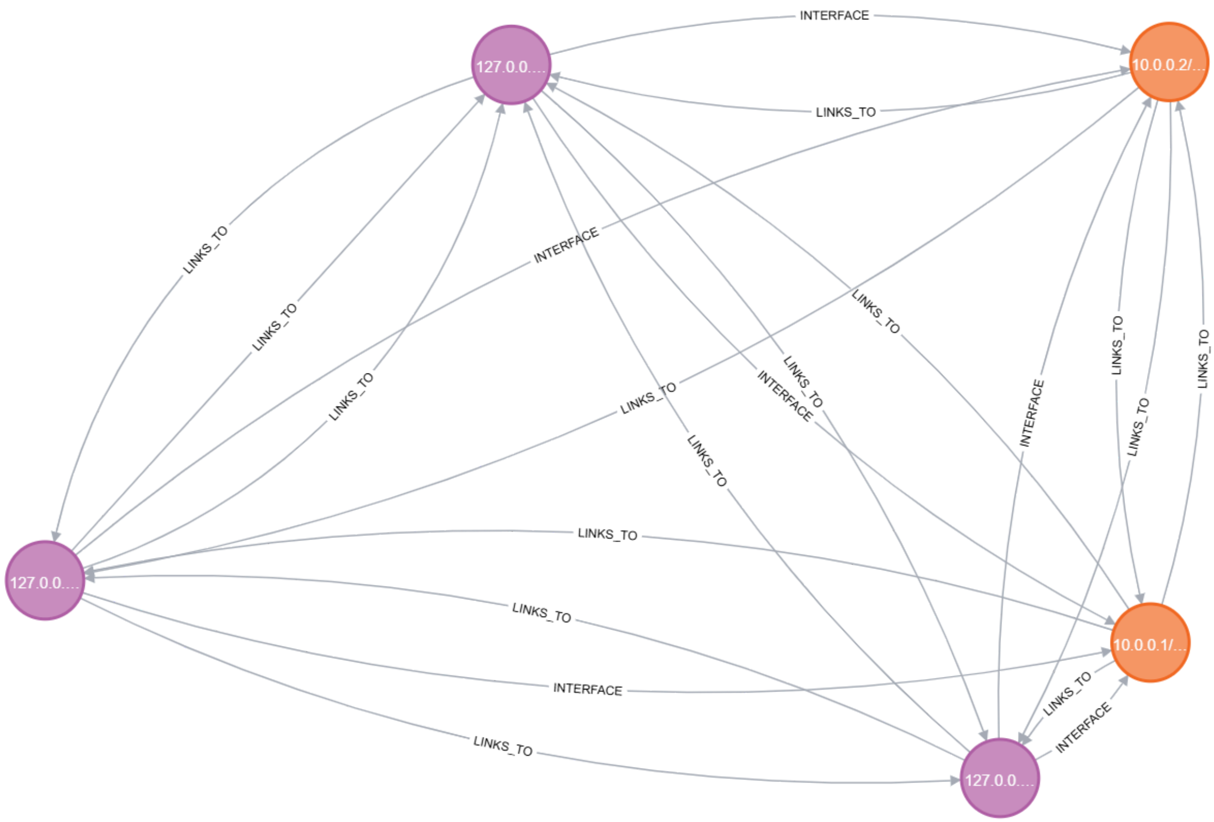
Query to list all possible paths between h1 and h2

MATCH p = allShortestPaths((h1:Host {id:'h1'})-[:LINKS_TO*..5]-(h2:Host {id:'h2'}))
RETURN p;



Query to list all paths up to 6 jumps (even if they are not the shortest)

MATCH p = (h1:Host {id:'h1'})-[:LINKS_TO*1..6]-(h2:Host {id:'h2'})
RETURN p;



Query to count the number of distinct paths

MATCH p = allShortestPaths((h1:Host {id:'h1'})-[:LINKS_TO*..6]-(h2:Host {id:'h2'}))
RETURN count(p) AS total_paths;

If total_paths > 1, it means that the network is in fact fault tolerant (there is redundancy).

Return was 2

Makes it more secure

Strategies to increase security in the graph

1. **Firewalls / ACLs (Access Control Lists)**
   - Create Firewall nodes or relationships [:ALLOWS] / [:DENIES] to indicate which hosts can communicate.
2. **Switch authentication**
   - Add properties such as certificate, public_key or auth_required: true to the switches.
3. **Network segmentation (VLANs or subnets)**
   - Create [:PART_OF {vlan: 10}] relationships between hosts and switches, ensuring that each group only talks to each other.
4. **Monitoring / IDS**
   - Create an :IDS or : Monitor node connected to the switches, representing traffic surveillance.

1. Create a firewall and define policies

```
// Create a firewall between h1 and h2
MERGE (fw:Firewall {id:'fw1', rules:'deny all; allow tcp/22,80,443'})
MERGE (fw)-[:PROTECTS]->(h1)
MERGE (fw)-[:PROTECTS]->(h2);
```

2. Require authentication on switches

```
MATCH (s:Switch)
SET s.auth_required = true,
   s.certificate = 'cert-switch-' + s.id,
   s.encryption = 'TLS';
```

3. Create VLANs to segment traffic

```
MATCH (h1:Host {id:'h1'}), (h2:Host {id:'h2'}), (s1:Switch {id:'s1'})
MERGE (h1)-[:PART_OF {vlan: 10}]->(s1)
MERGE (h2)-[:PART_OF {vlan: 20}]->(s1);
```

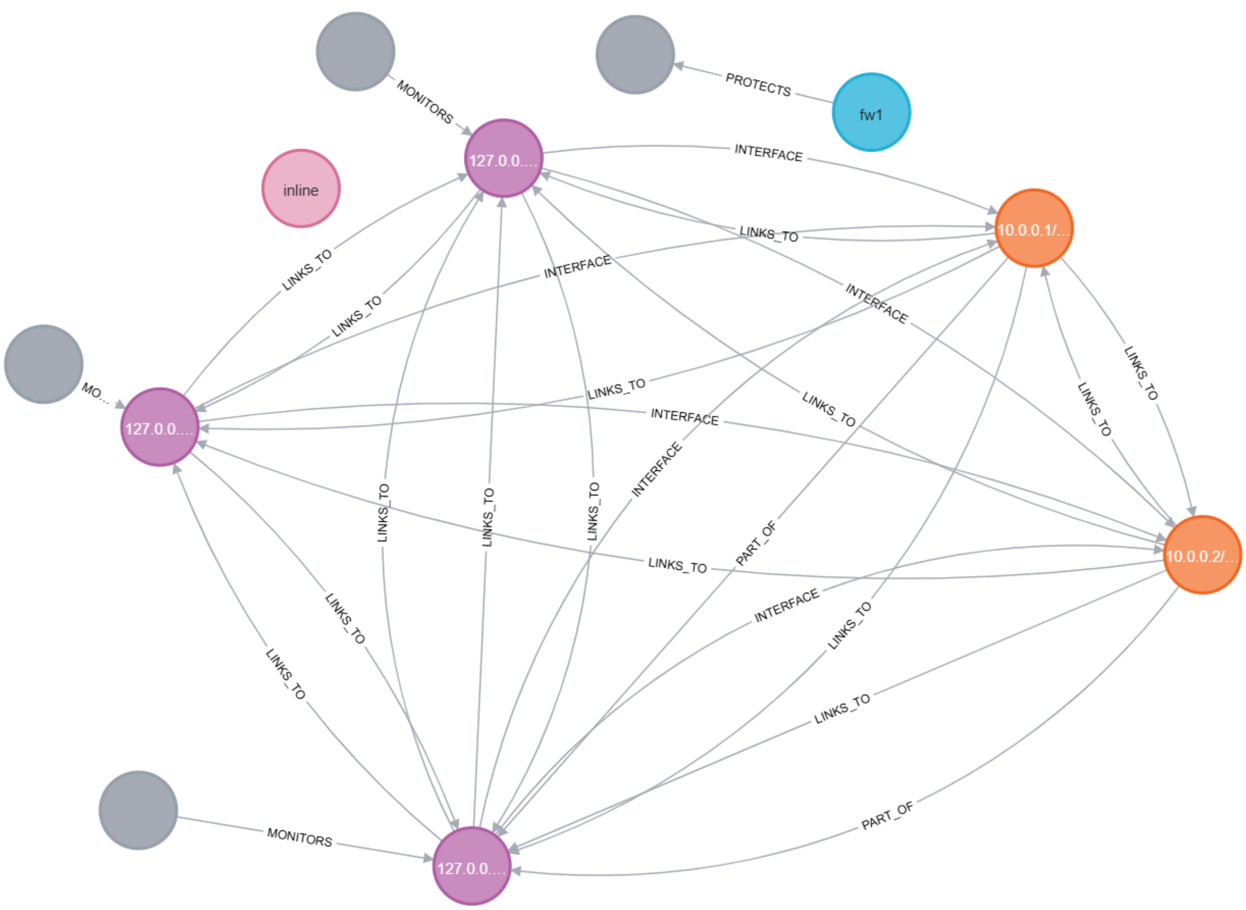
4. Central monitoring

Hosts are on segmented networks → less risk of lateral attack.

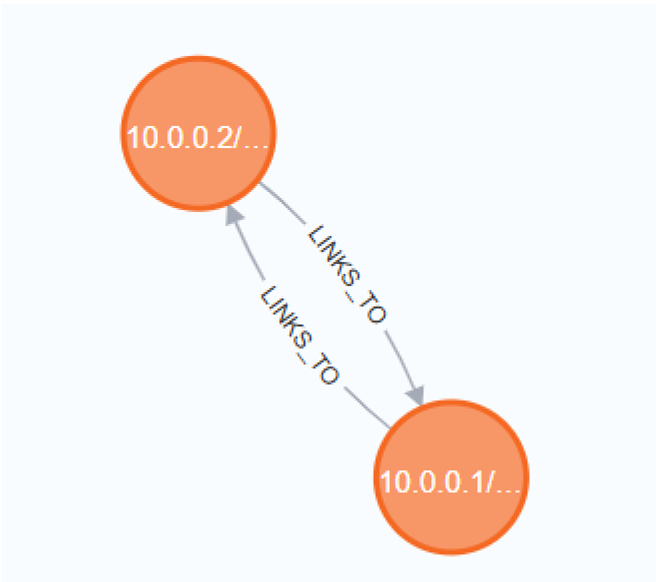
Switches require authentication and encryption.

There is a central firewall with explicit rules.

IDS ensures detection of malicious traffic.



See redundant paths between h1 and h2



See applied security