Load Balancer IP Pool Allocation

David Walter

2025-10-01

Contents

1 Overview **2**
1.1 Diagrams in This Document . . . 2

2 Architecture **3**
2.1 Network Architecture Diagram . . . 3
2.2 Connection Flow Diagram . . . 4
2.3 IP Pool Management . . . 4
2.4 Location in Codebase . . . 4

3 IP Allocation Process **5**
3.1 Initialization . . . 5
3.2 Service Binding Flow . . . 6
3.3 Code Flow . . . 6

4 IP Pool Implementation **8**
4.1 IP Lifecycle State Diagram . . . 8
4.2 Data Structure . . . 8
4.3 Allocation Algorithms . . . 9
4.4 Release Mechanism . . . 10

5 Error Detection **10**
5.1 Address Already In Use Detection . . . 10

6 CIDR Range Calculation **11**
6.1 Example: /28 CIDR (192.168.0.224/28) . . . 11
6.2 Supported CIDR Sizes . . . 11

7 Network Integration **11**
7.1 How Host Connects to k3d Pods . . . 11
7.2 Routing Configuration . . . 12
7.3 Connection Path . . . 12

8 TCP Proxying **12**
8.1 Bidirectional Pipe Implementation . . . 12

9 IP Sharing Between Services **12**
9.1 Overview . . . 12
9.2 Usage Examples . . . 13
9.3 Dynamic vs Static IP Allocation . . . 14
9.4 Limitations . . . 14

10 Configuration **14**
10.1 Command Line Options . . . 14
10.2 Docker Container Execution . . . 14
10.3 Required Capabilities . . . 15

11 Operational Behavior **15**
11.1 Service Creation Example . . . 15
11.2 Service Deletion Example . . . 15

12 Testing **15**
12.1 Manual Testing Steps . . . 15
12.2 Expected Results . . . 16

13 Troubleshooting **16**
13.1 IP Pool Exhaustion . . . 16
13.2 Address Already In Use (All IPs) . . . 16
13.3 Network Interface Not Found . . . 17

14 Future Enhancements **17**
14.1 Potential Improvements . . . 17

15 References **17**
15.1 Source Files . . . 17
15.2 Related Documentation . . . 17

16 Copyright **18**

1 Overview

The load balancer implements dynamic IP allocation from a CIDR range to support multiple Kubernetes services without manual IP assignment. When an IP address is already in use, the system automatically allocates the next available IP from the configured CIDR pool.

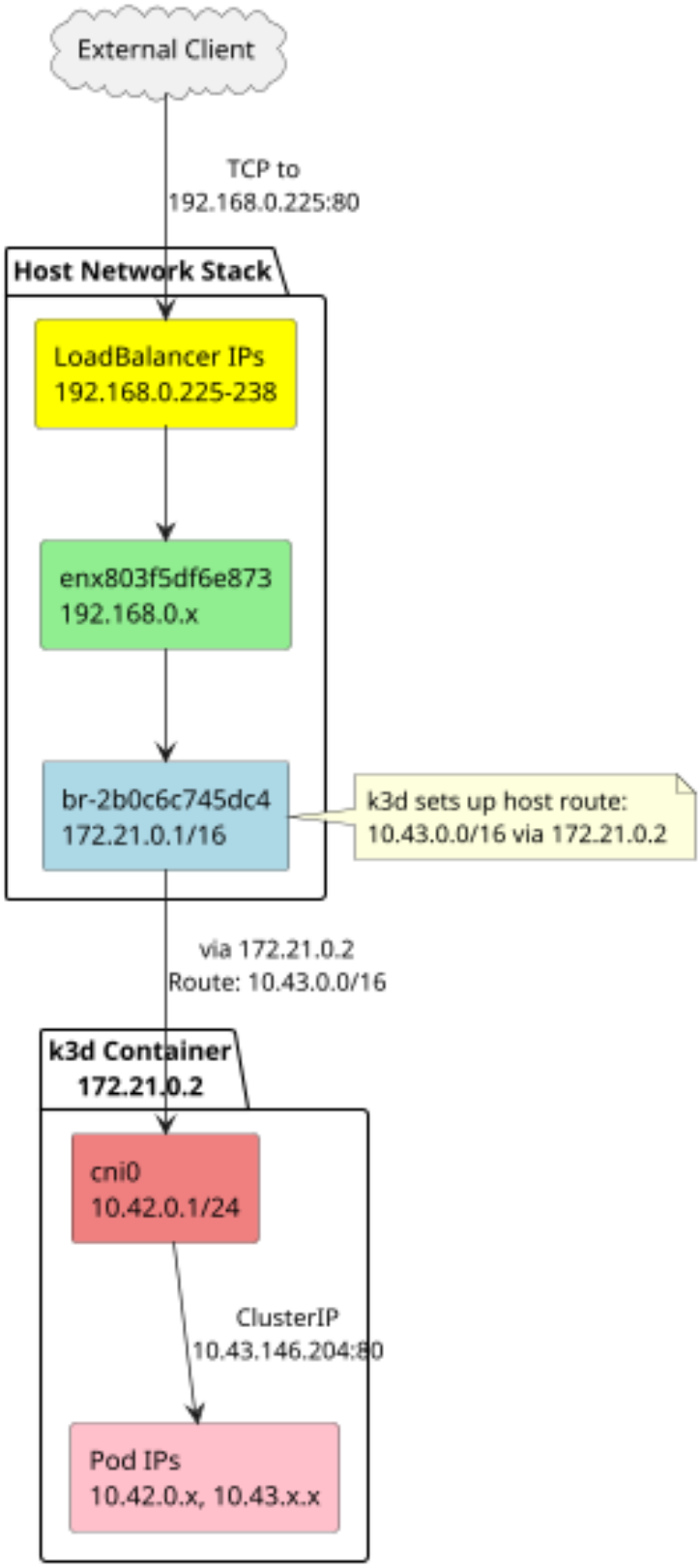
1.1 Diagrams in This Document

Diagram	File	Description
Network Architecture	network-architecture.png	Shows host/k3d network topology
Connection Flow	connection-flow.png	Sequence diagram of TCP proxy flow
Service Binding Flow	service-binding-flow.png	Activity diagram of IP allocation with retry
IP Lifecycle	ip-lifecycle.png	State diagram of IP allocation/release lifecycle

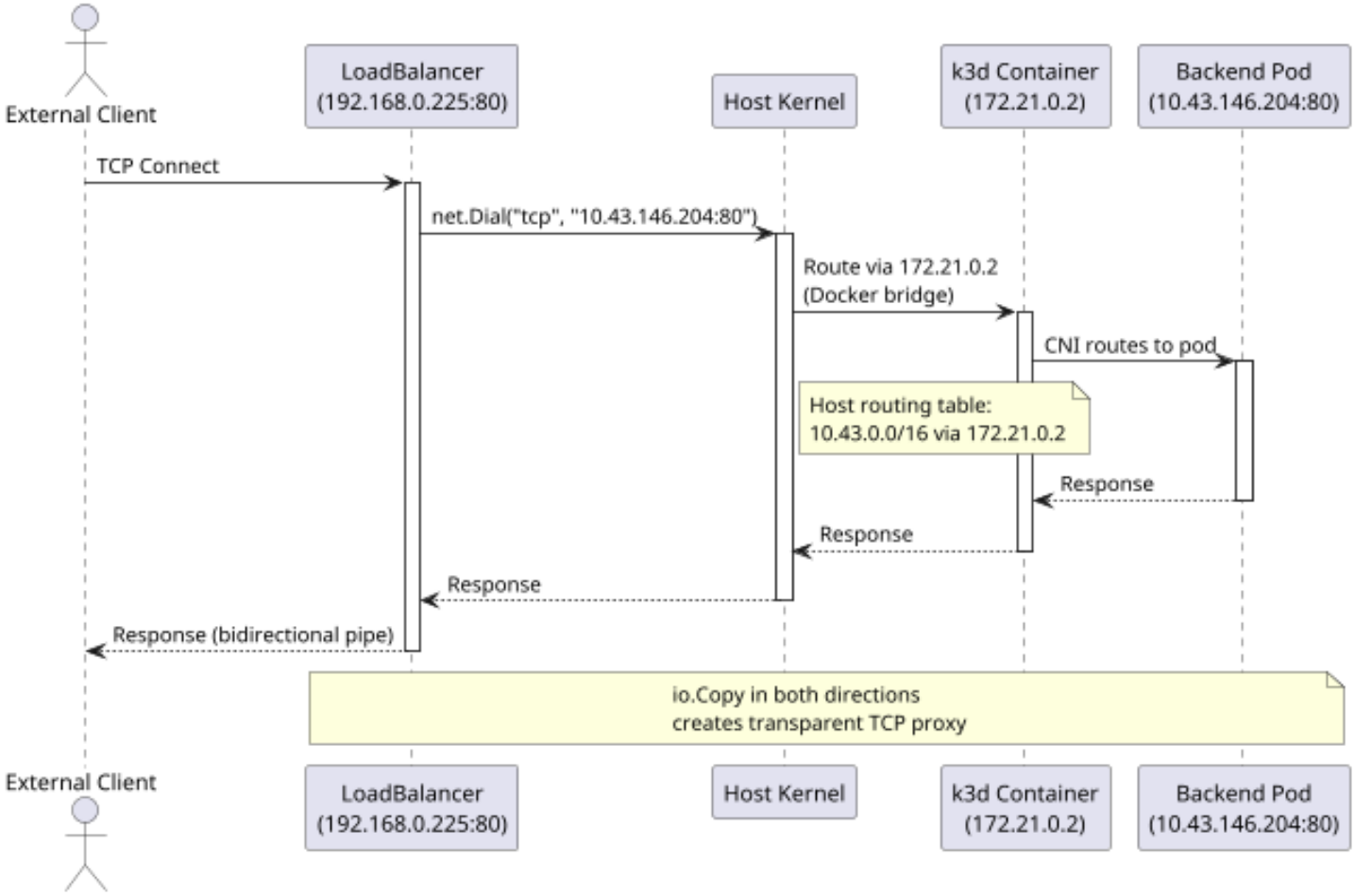
To generate diagrams, position cursor on each `#+begin_src plantuml` block and press `C-c C-c`.

2 Architecture

2.1 Network Architecture Diagram



2.2 Connection Flow Diagram



2.3 IP Pool Management

The IP pool manager maintains a thread-safe allocation table for IP:port combinations within a configured CIDR range. It tracks which IP addresses and ports are currently allocated, allowing multiple services to share the same IP on different ports.

2.3.1 Key Components

`IPPool` Thread-safe IP:port allocation manager

`IPPoolInstance` Global instance initialized from DefaultCIDR

`Allocate()` Returns next available IP from the pool (for dynamic allocation)

`AllocateSpecific(ip, port)` Allocates a specific IP:port combination

`Release(ip, port)` Returns IP:port to the pool for reuse

`RegisterPort(ip, port)` Registers a port for an already-allocated IP

2.4 Location in Codebase

`ipmgr/cidr.go:154-295` IPPool type and methods

`ipmgr/defaults.go:38-39` Global IPPoolInstance variable

`mgr/ipmgrinit.go:105-111` IP pool initialization

`mgr/open.go:83-193` IP allocation with retry logic and specific IP support

`mgr/listener.go:384-395` IP cleanup on listener close

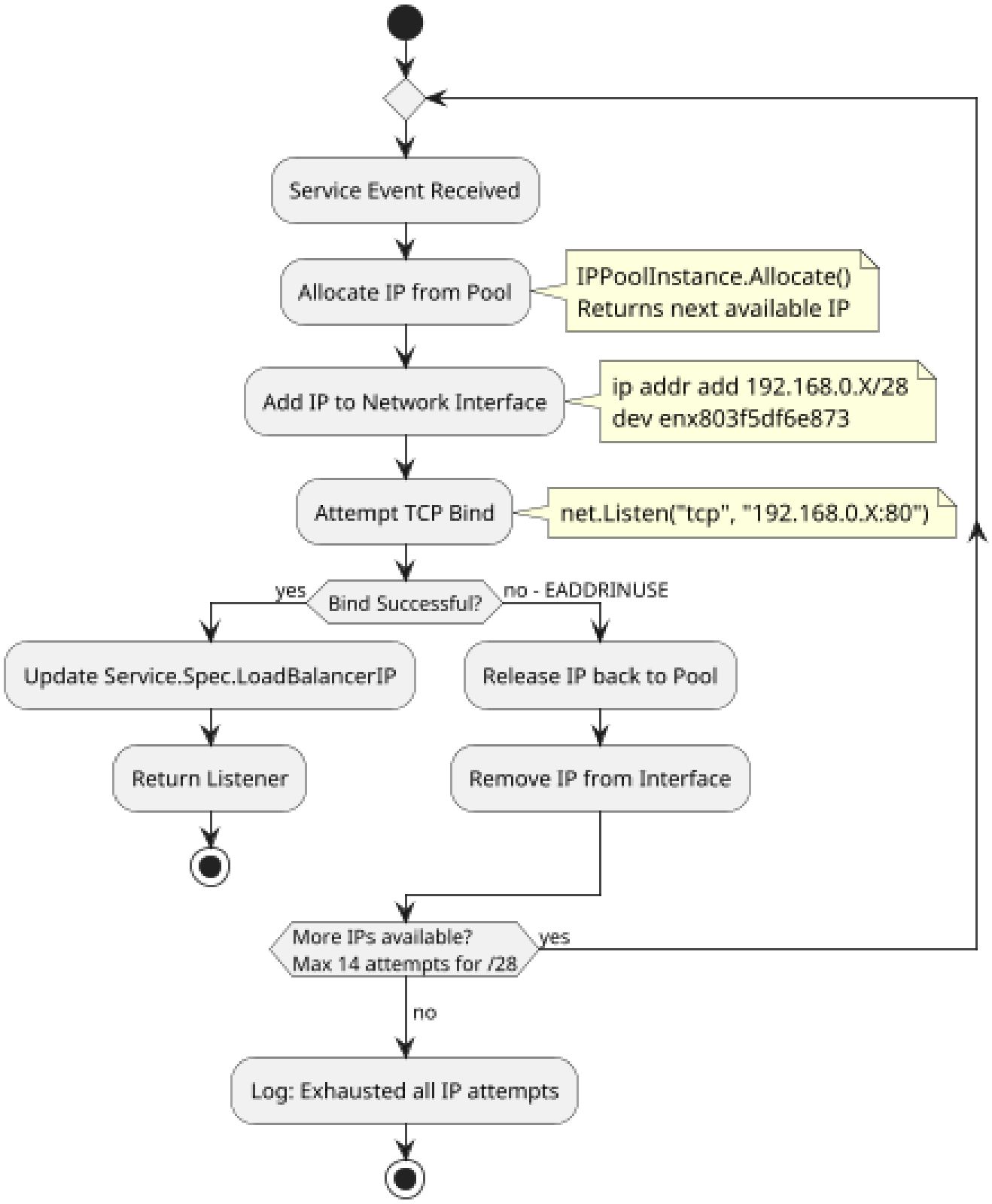
3 IP Allocation Process

3.1 Initialization

When the load balancer starts, it initializes the IP pool from the configured CIDR range:

```
// Initialize IP pool from the CIDR range
var err error
ipmgr.IPPoolInstance, err = ipmgr.NewIPPool(ipmgr.DefaultCIDR.String())
if err != nil {
    log.Fatalf("Failed to create IP pool from CIDR %s: %v",
               ipmgr.DefaultCIDR.String(), err)
}
log.Printf("Initialized IP pool for CIDR: %s", ipmgr.DefaultCIDR.String())
```

3.2 Service Binding Flow



3.3 Code Flow

3.3.1 ListenWithIPAllocation Function

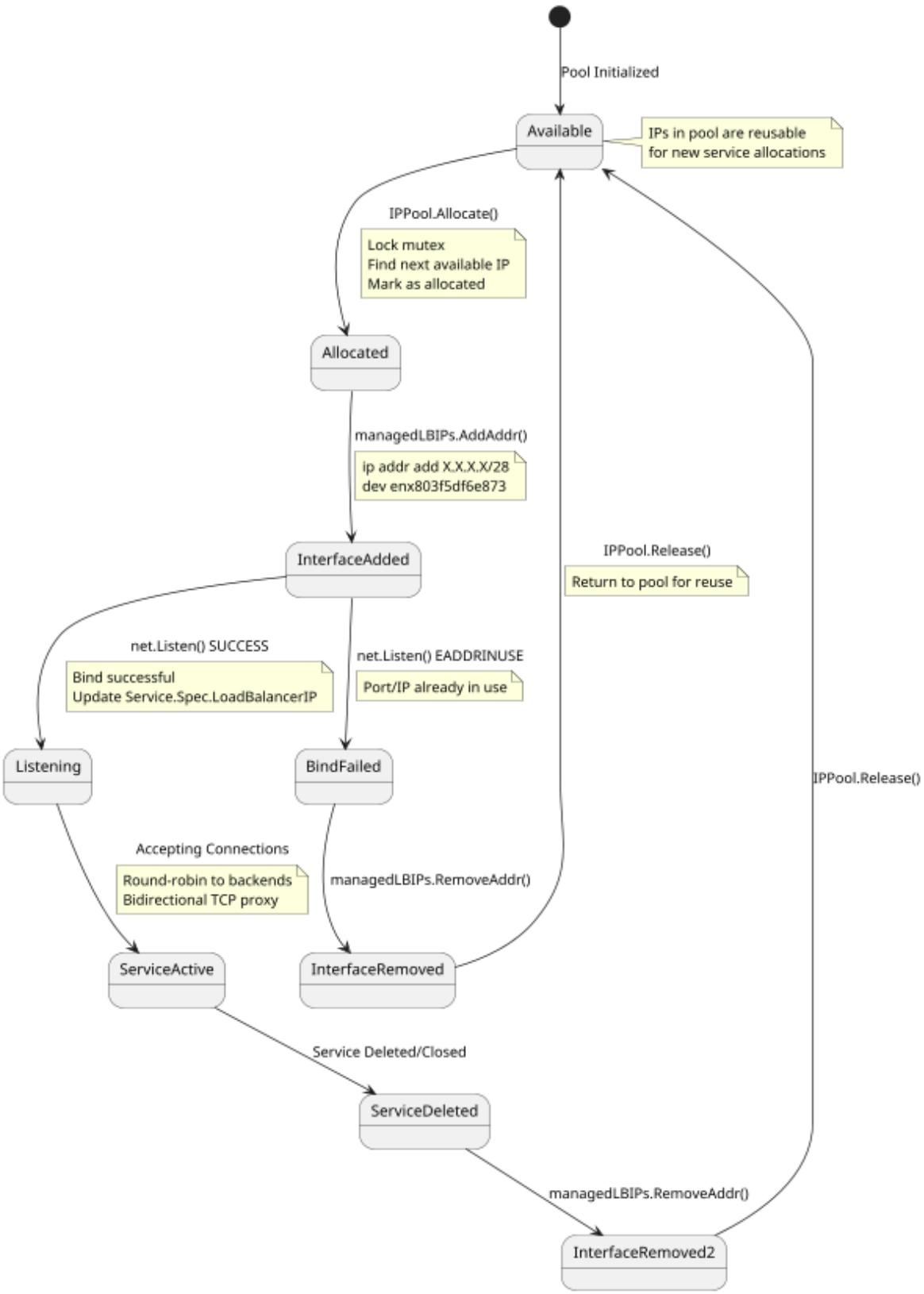
The `ListenWithIPAllocation` function in `mgr/open.go:83-153` implements the retry logic:

```
func ListenWithIPAllocation(serviceKey string, Service *v1.Service,
```

```
                        linkDevice string, cancel chan struct{}) (listener net.Listener) {

    maxIPAttempts := 14 // For /28 CIDR: 16 - network - broadcast

    for attempt := 0; attempt < maxIPAttempts; attempt++ {
        // 1. Allocate IP from pool
        allocatedIP, err := ipmgr.IPPoolInstance.Allocate()
        if err != nil {
            return nil
        }

        // 2. Add IP to network interface
        cidrStr := fmt.Sprintf("%s/%s", allocatedIP, ipmgr.Bits)
        managedLBIPs.AddAddr(cidrStr, linkDevice)

        // 3. Try to bind
        address := fmt.Sprintf("%s:%s", allocatedIP, port)
        listener, err = net.Listen("tcp", address)

        if err == nil {
            // Success!
            Service.Spec.LoadBalancerIP = allocatedIP
            return listener
        }

        // 4. Handle "address already in use"
        if isAddressInUse(err) {
            ipmgr.IPPoolInstance.Release(allocatedIP)
            managedLBIPs.RemoveAddr(cidrStr, linkDevice)
            continue // Try next IP
        }

        // Other errors are fatal
        return nil
    }

    return nil // Exhausted all IPs
}
```

4 IP Pool Implementation

4.1 IP Lifecycle State Diagram



4.2 Data Structure

```
type IPPool struct {
    cidr      *net.IPNet                  // CIDR range (e.g., 192.168.0.224/28)
```

```
    allocated map[string]bool             // IP address allocation tracking
    ports     map[string]map[string]bool  // IP -> port -> allocated (for port tracking)
    mu        sync.Mutex                  // Thread-safe access
}
```

This structure allows multiple services to share the same IP address on different ports. The `allocated` map tracks which IPs are in use, while the `ports` map tracks which specific ports are allocated on each IP.

4.3 Allocation Algorithms

4.3.1 Dynamic IP Allocation

The dynamic allocation algorithm iterates through all IPs in the CIDR range, skipping:

- Network address (first IP)
- Broadcast address (last IP)
- Already allocated IPs

```
func (p *IPPool) Allocate() (string, error) {
    p.mu.Lock()
    defer p.mu.Unlock()

    // Create a copy of the network IP to avoid modifying the original
    startIP := make(net.IP, len(p.cidr.IP))
    copy(startIP, p.cidr.IP)

    // Iterate through CIDR range
    for ip := startIP.Mask(p.cidr.Mask); p.cidr.Contains(ip); incIP(ip) {
        ipStr := ip.String()

        // Skip network and broadcast
        if p.isNetworkOrBroadcast(ip) {
            continue
        }

        // Check if available
        if !p.allocated[ipStr] {
            p.allocated[ipStr] = true
            // Initialize port map for this IP
            if p.ports[ipStr] == nil {
                p.ports[ipStr] = make(map[string]bool)
            }
            return ipStr, nil
        }
    }

    return "", fmt.Errorf("no available IPs in pool")
}
```

4.3.2 Specific IP:Port Allocation

When a service specifies `spec.loadBalancerIP`, the system allocates that specific IP with the service's port. Multiple services can share the same IP if they use different ports.

```
func (p *IPPool) AllocateSpecific(requestedIP, port string) error {
    p.mu.Lock()
    defer p.mu.Unlock()

    // Parse and validate the requested IP
    ip := net.ParseIP(requestedIP)
    if ip == nil {
```

```
        return fmt.Errorf("invalid IP address: %s", requestedIP)
    }

    // Check if IP is in the CIDR range
    if !p.cidr.Contains(ip) {
        return fmt.Errorf("IP %s is not in CIDR range %s", requestedIP, p.cidr.String())
    }

    // Check if it's a network or broadcast address
    if p.isNetworkOrBroadcast(ip) {
        return fmt.Errorf("IP %s is a network or broadcast address", requestedIP)
    }

    // Initialize port map for this IP if it doesn't exist
    if p.ports[requestedIP] == nil {
        p.ports[requestedIP] = make(map[string]bool)
    }

    // Check if this IP:port combination is already allocated
    if p.ports[requestedIP][port] {
        return fmt.Errorf("IP:port %s:%s is already allocated", requestedIP, port)
    }

    // Allocate the IP:port combination
    p.allocated[requestedIP] = true       // Mark IP as in use
    p.ports[requestedIP][port] = true     // Mark port as in use on this IP
    return nil
}
```

4.4 Release Mechanism

When a service is deleted or its listener closes, the IP:port combination is released back to the pool. If this is the last port using that IP, the IP itself is also freed.

```
func (p *IPPool) Release(ip, port string) {
    p.mu.Lock()
    defer p.mu.Unlock()

    // Release the specific port
    if p.ports[ip] != nil {
        delete(p.ports[ip], port)

        // If no more ports are allocated on this IP, free the IP entirely
        if len(p.ports[ip]) == 0 {
            delete(p.ports, ip)
            delete(p.allocated, ip)
        }
    }
}
```

This mechanism ensures that:

- Multiple services can share an IP on different ports
- IPs are only fully released when all ports are freed
- The network interface address remains until all services on that IP are deleted

5 Error Detection

5.1 Address Already In Use Detection

The `isAddressInUse` function detects bind failures due to port/IP conflicts:

```
func isAddressInUse(err error) bool {
    if err == nil {
        return false
    }

    // Check for syscall.EADDRINUSE
    if opErr, ok := err.(*net.OpError); ok {
        if osErr, ok := opErr.Err.(*syscall.Errno); ok {
            return *osErr == syscall.EADDRINUSE
        }
        return strings.Contains(opErr.Error(), "address already in use")
    }

    return strings.Contains(err.Error(), "address already in use")
}
```

6 CIDR Range Calculation

6.1 Example: /28 CIDR (192.168.0.224/28)

Purpose	Address	Notes
Network Address	192.168.0.224	Skipped (not allocated)
Usable IP 1	192.168.0.225	First allocatable IP
Usable IP 2	192.168.0.226	Second allocatable IP
…	…	…
Usable IP 14	192.168.0.238	Last allocatable IP
Broadcast	192.168.0.239	Skipped (not allocated)
Total IPs	16	
Usable IPs	14	Available for allocation

6.2 Supported CIDR Sizes

The implementation supports any valid IPv4 CIDR range:

`/28` 14 usable IPs (common for small deployments)

`/27` 30 usable IPs

`/26` 62 usable IPs

`/24` 254 usable IPs (typical class C network)

7 Network Integration

7.1 How Host Connects to k3d Pods

The load balancer running on the host can connect to k3d pod ClusterIPs through Docker's bridge network:

```
Host Network Stack
    br-2b0c6c745dc4 (172.21.0.1/16) ← Docker bridge for k3d-db
        k3d-db-server-0 (172.21.0.2) ← k3d container
            cni0 (10.42.0.1/24) ← CNI bridge inside container
                Pod IPs (10.42.0.x, 10.43.x.x ClusterIPs)

    enx803f5df6e873 (192.168.0.x) ← External interface
        192.168.0.225-238 ← LoadBalancer ExternalIPs
```

7.2 Routing Configuration

k3d automatically sets up host routes to enable pod connectivity:

```
# View routes to k3d networks
ip route | grep -E "172.21|10.43"

# Example output:
# 10.43.0.0/16 via 172.21.0.2 dev br-2b0c6c745dc4
# 172.21.0.0/16 dev br-2b0c6c745dc4 proto kernel scope link src 172.21.0.1
```

7.3 Connection Path

```
External Client
  ↓ TCP to 192.168.0.225:80 (ExternalIP on host interface)
LoadBalancer (listens on host with --network=host)
  ↓ Round-robin selects backend pod
  ↓ TCP Dial to 10.43.146.204:80 (pod ClusterIP)
Host Kernel
  ↓ Routes via 172.21.0.2 (k3d container)
k3d Container
  ↓ Routes to pod via CNI
Backend Pod
```

8 TCP Proxying

8.1 Bidirectional Pipe Implementation

The load balancer uses a bidirectional pipe pattern in `pipe/pipe.go:58-78` to forward traffic between client and backend:

```
func (pipe *Pipe) Connect() {
    done := make(chan bool, 2)

    // Client → Backend
    go func() {
        defer pipe.Close()
        io.Copy(pipe.SinkConn, pipe.SourceConn)
        done <- true
    }()

    // Backend → Client
    go func() {
        defer pipe.Close()
        io.Copy(pipe.SourceConn, pipe.SinkConn)
        done <- true
    }()
}
```

This creates a transparent TCP proxy that:

- Copies bytes in both directions simultaneously
- Doesn't parse or modify HTTP/TCP content
- Handles connection cleanup automatically

9 IP Sharing Between Services

9.1 Overview

The load balancer supports multiple services sharing the same external IP address on different ports. This is useful for:

- Conserving IP addresses in limited IP pools
- Hosting multiple protocols on the same IP (e.g., HTTP on port 80, HTTPS on 443)
- Grouping related services under a single IP

9.2 Usage Examples

9.2.1 Example 1: HTTP and HTTPS on Same IP

```
---
apiVersion: v1
kind: Service
metadata:
  name: web-http
spec:
  type: LoadBalancer
  loadBalancerIP: 192.168.0.226
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: webapp
---
apiVersion: v1
kind: Service
metadata:
  name: web-https
spec:
  type: LoadBalancer
  loadBalancerIP: 192.168.0.226  # Same IP, different port
  ports:
  - port: 443
    targetPort: 8443
  selector:
    app: webapp
```

9.2.2 Example 2: Multiple Protocols on Same IP

```
---
apiVersion: v1
kind: Service
metadata:
  name: app-http
spec:
  type: LoadBalancer
  loadBalancerIP: 192.168.0.227
  ports:
  - port: 8080
    targetPort: 8080
  selector:
    app: myapp
---
apiVersion: v1
kind: Service
metadata:
  name: app-grpc
spec:
  type: LoadBalancer
  loadBalancerIP: 192.168.0.227  # Same IP, different port
  ports:
  - port: 9090
```

```
    targetPort: 9090
  selector:
    app: myapp
```

9.3 Dynamic vs Static IP Allocation

9.3.1 Static IP Allocation (with IP sharing)

When `spec.loadBalancerIP` is specified:

- The load balancer attempts to bind to that specific IP:port combination
- Multiple services can use the same IP if they use different ports
- The IP:port combination must be unique
- Fails if the IP:port is already allocated

9.3.2 Dynamic IP Allocation (each service gets its own IP)

When `spec.loadBalancerIP` is NOT specified:

- The load balancer allocates the next available IP from the pool
- Each dynamically allocated service gets its own unique IP
- No IP sharing occurs with dynamic allocation

9.4 Limitations

- Cannot share the same IP:port combination (standard TCP limitation)
- All services sharing an IP must be in the CIDR range
- The network interface address is added once per IP (not per service)

10 Configuration

10.1 Command Line Options

```
# Run with specific CIDR range
bin/loadbalancer \
    --kubeconfig=/etc/kubernetes/config.k3d \
    --restricted-cidr=192.168.0.224/28 \
    --link-device=enx803f5df6e873 \
    --debug=true
```

10.2 Docker Container Execution

```
docker run --name loadbalancer \
    --cap-add=NET_ADMIN \
    --cap-add=NET_RAW \
    --cap-add=NET_BIND_SERVICE \
    --network=host \
    -e KUBERNETES=true \
    -v $HOME/.kube/config.k3d:/etc/kubernetes/config.k3d \
    -e KUBECONFIG=/etc/kubernetes/config.k3d \
    -e DEBUG=true \
    -e LINKDEVICE=enx803f5df6e873 \
    -e RESTRICTED_CIDR=192.168.0.224/28 \
    kdc2:5000/loadbalancer:latest
```

10.3 Required Capabilities

The load balancer requires these Linux capabilities to manage network interfaces:

CAP_NET_ADMIN Add/remove IP addresses on interfaces

CAP_NET_RAW Raw socket operations

CAP_NET_BIND_SERVICE Bind to privileged ports (<1024)

Set via `make setcap`:

```
sudo setcap 'cap_net_admin,cap_net_raw,cap_net_bind_service=+ep' bin/loadbalancer
```

11 Operational Behavior

11.1 Service Creation Example

```
2025/10/01 19:18:49 Initialized IP pool for CIDR: 192.168.0.224/28
2025/10/01 19:18:49 ServiceWatcher Event default/sample-service for type ADD
2025/10/01 19:18:49 Service default/sample-service listener create start
2025/10/01 19:18:49 Service default/sample-service Attempting to bind to 192.168.0.225:80 (attempt 1/14)
2025/10/01 19:18:49 Service default/sample-service Address 192.168.0.225:80 already in use, trying next IP
2025/10/01 19:18:49 Service default/sample-service Attempting to bind to 192.168.0.226:80 (attempt 2/14)
2025/10/01 19:18:49 Service default/sample-service Successfully bound to 192.168.0.226:80
2025/10/01 19:18:49 SetExternalIP [192.168.0.226]
2025/10/01 19:18:49 Service default/sample-service listener created 192.168.0.226:80
```

11.2 Service Deletion Example

```
2025/10/01 19:20:15 Shutting down listener for default/sample-service
2025/10/01 19:20:15 Released IP 192.168.0.226 back to pool for service default/sample-service
2025/10/01 19:20:15 Removing ExternalIP [192.168.0.226]
2025/10/01 19:20:15 RemoveAddr 192.168.0.226/28 enx803f5df6e873
```

12 Testing

12.1 Manual Testing Steps

1. Start the load balancer:

   ```
   bin/loadbalancer --kubeconfig=$HOME/.kube/config.k3d --debug=true
   ```

2. Create a test service in k3d:

   ```
   kubectl apply -f - <<EOF
   apiVersion: v1
   kind: Service
   metadata:
     name: test-service-1
   spec:
     type: LoadBalancer
     ports:
     - port: 80
       targetPort: 8080
     selector:
       app: test
   EOF
   ```

3. Verify IP allocation:

```
# Check assigned external IP
kubectl get svc test-service-1 -o jsonpath='{.spec.externalIPs[0]}'

# Verify IP is on network interface
ip addr show enx803f5df6e873 | grep 192.168.0
```

4. Create a second service to test IP allocation:

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Service
metadata:
  name: test-service-2
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: test
EOF
```

5. Verify both services have different IPs:

```
kubectl get svc -o wide | grep test-service
```

12.2 Expected Results

- First service gets `192.168.0.225` (or next available)
- Second service gets `192.168.0.226` (next in sequence)
- Deleting first service releases `192.168.0.225` back to pool
- Creating third service reuses `192.168.0.225`

13 Troubleshooting

13.1 IP Pool Exhaustion

If all IPs in the CIDR are allocated:

```
ERROR: Failed to allocate IP from pool: no available IPs in pool
```

Solution: Increase CIDR size or delete unused services.

13.2 Address Already In Use (All IPs)

If all attempts fail:

```
ERROR: Service default/test-service Exhausted all IP allocation attempts
```

Possible Causes:

- External process using IPs in the range
- Another load balancer instance running
- Stale IPs from previous crashes

Solution:

```
# Check what's listening on the IPs
for ip in {225..238}; do
    sudo ss -tlnp | grep "192.168.0.$ip:80"
done

# Manually release stuck IPs
sudo ip addr del 192.168.0.225/28 dev enx803f5df6e873
```

13.3 Network Interface Not Found

```
ERROR: Auto-detected interface 'enx803f5df6e873' also has no IPv4 addresses
```

Solution: Verify interface exists and has IP:

```
ip addr show enx803f5df6e873
```

14 Future Enhancements

14.1 Potential Improvements

1. **Persistent IP Allocation**
   - Store IP allocations in Kubernetes annotations
   - Survive load balancer restarts with same IP assignments
2. **IP Affinity**
   - Prefer same IP when service is recreated
   - Use service name hash for deterministic allocation
3. **Multi-CIDR Support**
   - Support multiple CIDR ranges
   - Automatic failover to secondary CIDR when primary exhausted
4. **Health-Based Deallocation**
   - Detect and release IPs from services with no healthy endpoints
   - Automatic cleanup of orphaned IPs
5. **Metrics and Monitoring**
   - Expose IP pool utilization metrics
   - Alert when pool usage exceeds threshold

15 References

15.1 Source Files

- ipmgr/cidr.go:154 - IPPool implementation
- mgr/open.go:83 - ListenWithIPAllocation function
- mgr/ipmgrinit.go:105 - IP pool initialization
- mgr/listener.go:384 - IP cleanup on close

15.2 Related Documentation

- Docker Registry Setup
- Project README

16 Copyright

Copyright 2018-2025 David Walter.
Licensed under the Apache License, Version 2.0.