Network Discovery Technical Guide

# Network Discovery Technical Guide

## Multi-Protocol Discovery Implementation

Modern asyncio-based network discovery supporting mDNS, SSDP, and ARP scanning with concurrent execution patterns.

## mDNS/DNS-SD Discovery

### Core Implementation

```python
import asyncio
from zeroconf.asyncio import AsyncServiceBrowser, AsyncZeroconf

class MDNSDiscovery:
    """High-performance mDNS discovery with resource management."""

    def __init__(self, max_concurrent: int = 50) -> None:
        self.semaphore = asyncio.Semaphore(max_concurrent)
        self.service_types = ["_mcp._tcp.local."]
        self.discovered_services: dict[str, MCPServiceRecord] = {}

    async def discover_services(self, timeout: int = 30) -> list[MCPServiceRecord]:
        """Discover MCP servers using mDNS with timeout control."""
        aiozc = AsyncZeroconf()

        try:
            browser = AsyncServiceBrowser(
                aiozc.zeroconf,
                self.service_types,
                handlers=[self._on_service_state_change]
            )

            await asyncio.sleep(timeout)
            await browser.async_cancel()

            return list(self.discovered_services.values())
        finally:
            await aiozc.async_close()

    def _on_service_state_change(
        self,
        zeroconf: Zeroconf,
        service_type: str,
        name: str,
        state_change: str
    ) -> None:
        """Handle service discovery state changes."""
        if state_change == "Added":
            info = zeroconf.get_service_info(service_type, name)
            if info:
                server = self._create_server_from_info(info)
                self.discovered_services[server.id] = server
```

### Service Record Processing

```python
def _create_server_from_info(self, info: ServiceInfo) -> MCPServiceRecord:
    """Convert Zeroconf service info to MCP server record."""

    # Extract TXT record properties
    properties = {
        key.decode(): value.decode()
        for key, value in info.properties.items()
    }

    return MCPServiceRecord(
        id=f"mdns-{info.name}",
        name=properties.get("name", info.name),
        endpoint=f"http://{socket.inet_ntoa(info.addresses[0])}:{info.port}",
        version=properties.get("version", "1.0"),
        capabilities=properties.get("capabilities", "").split(","),
        auth_method=properties.get("auth", "none"),
        discovery_method="mdns"
    )
```

## SSDP/UPnP Discovery

### Windows Compatibility Implementation

```python
import socket
import select
from urllib.parse import urlparse

class SSDPDiscovery:
    """SSDP discovery for enterprise Windows networks."""

    MULTICAST_IP = "239.255.255.250"
    MULTICAST_PORT = 1900

    def __init__(self) -> None:
        self.search_target = "urn:schemas-mcp-org:device:MCPServer:1"
        self.discovered_services: dict[str, MCPServiceRecord] = {}

    async def discover_upnp_devices(self, timeout: int = 5) -> list[MCPServiceRecord]:
        """Discover MCP servers via SSDP multicast."""

        sock = self._create_multicast_socket()

        try:
            # Send M-SEARCH request
            search_message = self._build_search_message(timeout)
            sock.sendto(search_message.encode(), (self.MULTICAST_IP, self.MULTICAST_PORT))

            # Listen for responses
            await self._listen_for_responses(sock, timeout)

            return list(self.discovered_services.values())
        finally:
            sock.close()

    def _build_search_message(self, max_wait: int) -> str:
        """Build SSDP M-SEARCH message."""
        return (
            f'M-SEARCH * HTTP/1.1\r
'
            f'HOST: {self.MULTICAST_IP}:{self.MULTICAST_PORT}\r
'
            f'MAN: "ssdp:discover"\r
'
            f'MX: {max_wait}\r
'
            f'ST: {self.search_target}\r
'
            f'\r
'
        )
```

## Concurrent Network Scanning

### Resource-Controlled Implementation

```
class ConcurrentScanner:
    """Async network scanner with semaphore-based resource control."""

    def __init__(self, max_workers: int = 50, timeout: float = 5.0) -> None:
        self.semaphore = asyncio.Semaphore(max_workers)
        self.timeout = timeout
        self.connector = aiohttp.TCPConnector(
            limit=100,
            limit_per_host=30,
            ttl_dns_cache=300
        )

    async def scan_network_range(self, network: str) -> list[MCPServiceRecord]:
        """Scan IP range with controlled concurrency."""
        import ipaddress

        network_obj = ipaddress.IPv4Network(network, strict=False)

        # Create scan tasks for all hosts
        tasks = [
            asyncio.create_task(self._scan_host_with_semaphore(str(ip)))
            for ip in network_obj.hosts()
        ]

        # Execute with exception handling
        results = await asyncio.gather(*tasks, return_exceptions=True)

        # Filter successful results
        return [
            result for result in results
            if isinstance(result, MCPServiceRecord)
        ]

    async def _scan_host_with_semaphore(self, host: str) -> MCPServiceRecord | None:
        """Scan single host with semaphore protection."""
        async with self.semaphore:
            return await self._probe_mcp_server(host)
```

## Performance Optimization

### Discovery Caching

```python
from typing import Generic, TypeVar
import time

T = TypeVar("T")

class TTLCache(Generic[T]):
    """Time-to-live cache for discovery results."""

    def __init__(self, default_ttl: int = 300) -> None:
        self.cache: dict[str, CacheEntry[T]] = {}
        self.default_ttl = default_ttl

    async def get_or_compute(
        self,
        key: str,
        compute_func: Callable[[], Awaitable[T]],
        ttl: int | None = None
    ) -> T:
        """Get cached value or compute new one."""

        entry = self.cache.get(key)
        if entry and not entry.is_expired():
            return entry.data

        # Cache miss - compute new value
        data = await compute_func()
        self.cache[key] = CacheEntry(
            data=data,
            expires_at=time.time() + (ttl or self.default_ttl)
        )

        return data
```

### Connection Pooling

```python
class DiscoveryClient:
    """HTTP client with optimized connection pooling."""

    def __init__(self) -> None:
        self.connector = aiohttp.TCPConnector(
            limit=100,              # Total pool size
            limit_per_host=30,      # Per-host limit
            ttl_dns_cache=300,      # DNS cache TTL
            use_dns_cache=True,
            enable_cleanup_closed=True
        )

        self.session = aiohttp.ClientSession(
            connector=self.connector,
            timeout=aiohttp.ClientTimeout(total=30),
            headers={"User-Agent": "MCP-Vacuum/1.0"}
        )

    async def probe_server(self, endpoint: str) -> MCPCapabilities | None:
        """Probe MCP server capabilities with connection reuse."""
        try:
            async with self.session.get(f"{endpoint}/capabilities") as response:
                if response.status == 200:
                    data = await response.json()
                    return MCPCapabilities.parse_obj(data)
        except Exception:
            return None
```

## Discovery Constraints

### Network Security Implementation

```python
import ipaddress

class SecureDiscovery:
    """Discovery with network security constraints."""

    def __init__(self, allowed_networks: list[str]) -> None:
        self.allowed_networks = [
            ipaddress.IPv4Network(net, strict=False)
            for net in allowed_networks
        ]

    def is_allowed_host(self, host: str) -> bool:
        """Check if host is in allowed networks."""
        try:
            host_ip = ipaddress.IPv4Address(host)
            return any(host_ip in network for network in self.allowed_networks)
        except ValueError:
            return False

    async def secure_discovery(self) -> list[MCPServiceRecord]:
        """Perform discovery with security filtering."""
        all_discovered = await self.discovery_engine.scan_all_protocols()

        return [
            server for server in all_discovered
            if self.is_allowed_host(server.host)
        ]
```

## Error Handling & Resilience

### Circuit Breaker Pattern

```python
class DiscoveryCircuitBreaker:
    """Circuit breaker for discovery operations."""

    def __init__(self, failure_threshold: int = 5, timeout: int = 60) -> None:
        self.failure_threshold = failure_threshold
        self.timeout = timeout
        self.failure_count = 0
        self.last_failure_time = 0
        self.state = "closed"  # closed, open, half_open

    async def call(self, func: Callable[[], Awaitable[T]]) -> T:
        """Execute function with circuit breaker protection."""

        if self.state == "open":
            if time.time() - self.last_failure_time > self.timeout:
                self.state = "half_open"
            else:
                raise CircuitBreakerOpenError("Circuit breaker is open")

        try:
            result = await func()

            if self.state == "half_open":
                self.state = "closed"
                self.failure_count = 0

            return result

        except Exception as e:
            self.failure_count += 1
            self.last_failure_time = time.time()

            if self.failure_count >= self.failure_threshold:
                self.state = "open"

            raise e
```

## Next Steps

1. **Implement mDNS**: Start with basic mDNS discovery using zeroconf
2. **Add SSDP Support**: Implement SSDP for Windows compatibility
3. **Optimize Performance**: Add connection pooling and caching
4. **Security Integration**: Implement network constraints and filtering