Guida Completa: Service Discovery in Reti Locali

Indice

Introduzione al Problema

Il Problema

In un'applicazione client-server tradizionale, il client deve conoscere a priori l'indirizzo IP del server. Questo approccio presenta diversi limiti:

- **Rigidità**: ogni volta che il server cambia macchina, bisogna modificare il codice client
- **Manutenzione**: in ambienti con più server, gestire gli IP diventa complesso
- **Scalabilità**: difficile distribuire l'applicazione su diverse reti
- **Usabilità**: l'utente finale deve conoscere dettagli tecnici (IP, porta)

La Soluzione: Service Discovery

Il **Service Discovery** è un meccanismo che permette ai client di trovare automaticamente i servizi disponibili sulla rete senza configurazione manuale.

Requisiti base:

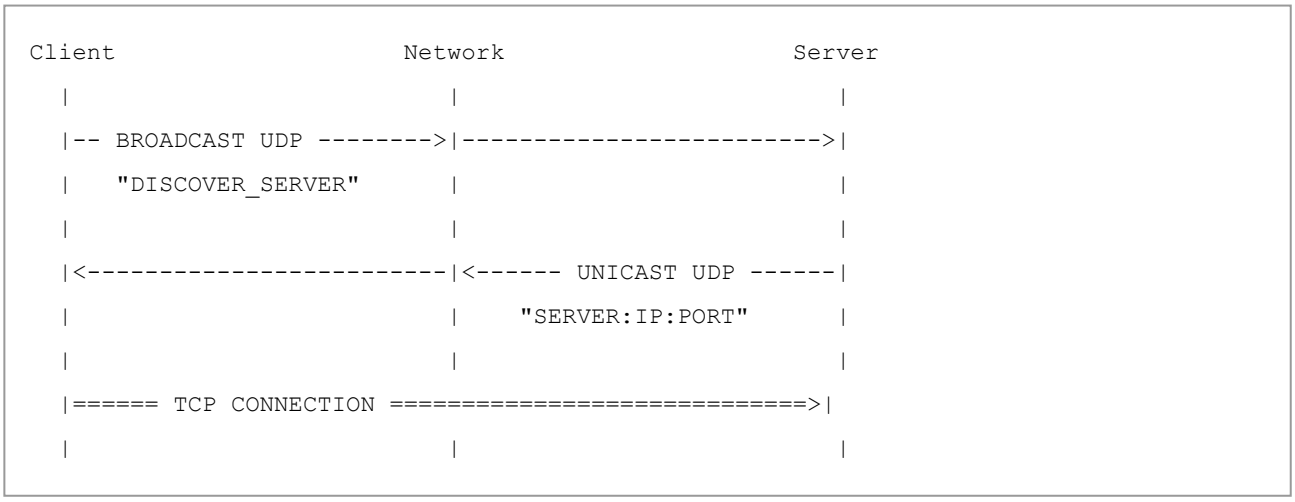
- Automatico (nessun intervento umano)
- Affidabile (deve trovare il server se esiste)
- Veloce (tempi di discovery ragionevoli)
- Semplice da implementare

Soluzione 1: UDP Broadcast Discovery

Concetto

Il client invia un messaggio **broadcast** UDP sulla rete locale. Tutti i dispositivi ricevono il messaggio, ma solo il server risponde identificandosi.

Architettura



Implementazione Dettagliata

Server - Servizio Discovery

```
def discovery_service(self):
    # Socket UDP per ricevere broadcast
    discovery_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    discovery_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    # Bind su TUTTE le interfacce di rete
    discovery_socket.bind(('', self.discovery_port))

    while self.running:
        data, addr = discovery_socket.recvfrom(1024)
        message = data.decode('utf-8')

        if message == "DISCOVER_CHAT_SERVER":
            # Risponde con IP e porta del servizio chat
            response = f"CHAT_SERVER:{self.get_local_ip()}:{self.chat_port}"
            discovery_socket.sendto(response.encode('utf-8'), addr)
```

Punti chiave:

- `SO_REUSEADDR`: permette di riutilizzare la porta anche se non completamente chiusa
- `bind(('', port))`: ascolta su tutte le interfacce di rete (0.0.0.0)
- Risposta unicast: torna solo al client richiedente

Client - Discovery

```
def discover_server(self, timeout=5):
    discovery_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # Abilita il broadcast
    discovery_socket.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
    discovery_socket.settimeout(timeout)

    # Invia broadcast a TUTTI i dispositivi nella subnet
    discovery_socket.sendto(
        "DISCOVER_CHAT_SERVER".encode('utf-8'),
        ('<broadcast>', self.discovery_port)
    )

    # Attende risposta
    data, addr = discovery_socket.recvfrom(1024)
    # Parsing della risposta...
```

Punti chiave:

- `SO_BROADCAST`: necessario per inviare pacchetti broadcast
- `<broadcast>`: indirizzo speciale 255.255.255.255
- `settimeout()`: evita attese infinite se nessun server risponde

Pro e Contro

□ Vantaggi

1. **Velocità eccezionale**
    - Discovery quasi istantaneo (< 1 secondo)
    - Un solo pacchetto broadcast necessario
    - Nessuna iterazione su IP multipli
2. **Efficienza di rete**
    - Traffico minimo generato
    - Un solo pacchetto broadcast + una risposta unicast

- Non sovraccarica la rete

3. **Scalabilità**

   - Funziona con qualsiasi numero di dispositivi
   - Costo computazionale costante O(1)
   - Non dipende dalla dimensione della subnet

4. **Semplicità concettuale**

   - Logica diretta e intuitiva
   - Facile debugging con Wireshark
   - Pattern standard nell'industria

5. **Supporto multipli server**

   - Più server possono rispondere contemporaneamente
   - Il client può scegliere o usare tutti i server trovati
   - Utile per load balancing o fault tolerance

6. **Didattico**

   - Insegna la differenza tra UDP e TCP
   - Mostra l'uso del broadcast
   - Introduce concetti di service discovery

□ Svantaggi

1. **Limitazioni di rete**

   - **Non attraversa i router**: funziona solo nella subnet locale (broadcast domain)
   - **Segmentazione VLAN**: non funziona tra VLAN diverse
   - **VPN**: problemi con alcune configurazioni VPN

2. **Firewall e sicurezza**

   - Molti firewall bloccano broadcast UDP per default
   - Windows Firewall può richiedere regole esplicite
   - Alcuni antivirus segnalano come sospetto

3. **Affidabilità UDP**

   - UDP non garantisce consegna (unreliable protocol)
   - Pacchetti possono essere persi senza notifica
   - Nessun acknowledgment o retry automatico

4. **Potenziali collisioni**

   - Se più server rispondono simultaneamente, possibili conflitti
   - Necessita logica di gestione risposte multiple

- Possibile confusione con servizi sulla stessa porta

5. **Congestione broadcast**

   - In reti molto grandi, il broadcast aggiunge traffico
   - Ogni dispositivo deve processare il pacchetto
   - Switch possono limitare broadcast rate

6. **Problemi con Wi-Fi**

   - Access Point possono filtrare broadcast
   - Reti guest spesso bloccano broadcast tra client
   - Modalità risparmio energetico può perdere pacchetti

Quando Usare UDP Broadcast

 Ideale per:

- Reti locali piccole/medie (< 254 host)
- Ambienti controllati (lab, ufficio)
- Applicazioni dove velocità è prioritaria
- Prototipi e proof of concept
- Contesto didattico (insegna networking)

 Evitare quando:

- Attraversamento router necessario
- Reti enterprise con policy rigide
- Ambienti con firewall restrittivi
- Necessità di garanzie di consegna

Varianti e Ottimizzazioni

Multicast invece di Broadcast

```
# Server
MCAST_GRP = '224.1.1.1'
sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP,
                socket.inet_aton(MCAST_GRP) + socket.inet_aton('0.0.0.0'))

# Client
sock.sendto(message, (MCAST_GRP, port))
```

Vantaggi del Multicast:

- Più efficiente del broadcast (solo interessati ricevono)

- Può attraversare router se configurato (con IGMP)
- Minore impatto sulla rete

Svantaggi:

- Configurazione più complessa
- Non sempre supportato da router consumer
- Richiede setup IGMP

Retry e Timeout Adattivo

```
def discover_server_with_retry(self, max_attempts=3):
    for attempt in range(max_attempts):
        timeout = 2 * (attempt + 1)  # Timeout crescente
        if self.discover_server(timeout):
            return True
    return False
```

Autenticazione e Sicurezza

```
import hmac
import hashlib

# Server
def verify_request(self, data, shared_secret):
    message, signature = data.split('|')
    expected = hmac.new(shared_secret, message.encode(), hashlib.sha256).hexdigest()
    return hmac.compare_digest(signature, expected)
```

Soluzione 2: Scansione Diretta della Subnet

Concetto

Il client determina la subnet locale (es. 192.168.1.0/24) e tenta di connettersi sequenzialmente o in parallelo a ogni IP sulla porta del server.

Architettura

```
Client                    Network
  |
  |-- Calcola subnet: 192.168.1.0/24
  |
  |-- TCP SYN --> 192.168.1.1:5000  (timeout/refused)
  |-- TCP SYN --> 192.168.1.2:5000  (timeout/refused)
  |-- TCP SYN --> 192.168.1.3:5000  (SUCCESS!) ✓
  |
  |====== Connessione stabilita ======>
```

Implementazione Dettagliata

Determinazione Subnet

```
def get_local_network(self):
    # Ottiene IP locale
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect(("8.8.8.8", 80))  # Non invia realmente dati
    local_ip = s.getsockname()[0]
    s.close()

    # Crea oggetto rete con subnet /24
    network = ipaddress.IPv4Network(f"{local_ip}/24", strict=False)
    # Es: 192.168.1.100 -> 192.168.1.0/24

    return network
```

Punti chiave:

- Trucco con `connect()` a 8.8.8.8: determina quale interfaccia userebbe il sistema
- `ipaddress.IPv4Network`: modulo standard Python per gestione IP
- `strict=False`: accetta IP host e calcola network address
- `/24`: subnet mask 255.255.255.0 (254 host utilizzabili)

Scansione con Threading

```
def scan_network(self, max_workers=50):
    network = self.get_local_network()
    found_servers = []

    with concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as executor:
        # Crea un future per ogni IP
        future_to_ip = {
            executor.submit(self.check_server, ip): ip
            for ip in network.hosts()
        }

        # Processa risultati man mano che arrivano
        for future in concurrent.futures.as_completed(future_to_ip):
            ip = future_to_ip[future]
            if future.result():
                found_servers.append(str(ip))

    return found_servers
```

Punti chiave:

- `ThreadPoolExecutor`: pool di thread per esecuzione parallela
- `max_workers=50`: numero massimo thread simultanei (bilanciamento carico/velocità)
- `as_completed()`: processa risultati appena disponibili (non attende tutti)
- `network.hosts()`: esclude network address e broadcast (x.x.x.0 e x.x.x.255)

Verifica Server

```
def check_server(self, ip, timeout=0.5):
    try:
        test_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        test_socket.settimeout(timeout)

        # connect_ex() ritorna 0 se successo, errno altrimenti
        result = test_socket.connect_ex((str(ip), self.server_port))

        test_socket.close()
        return result == 0  # True se connessione riuscita
    except:
        return False
```

Punti chiave:

- `connect_ex()`: versione non-blocking di connect() che ritorna errno
- Timeout breve (0.5s): bilancia velocità e affidabilità
- Gestione eccezioni: host irraggiungibili possono causare exception

Pro e Contro

□ Vantaggi

1. **Affidabilità massima**
    - Funziona sempre se il server è raggiungibile via TCP
    - Non dipende da broadcast o multicast
    - Nessuna dipendenza da configurazioni di rete speciali
2. **Attraversamento ostacoli**
    - Funziona con firewall che bloccano broadcast
    - Compatibile con reti Wi-Fi guest (isolamento client)
    - Nessun problema con policy di rete restrittive
3. **Compatibilità universale**
    - Funziona su qualsiasi rete TCP/IP
    - Nessuna configurazione speciale richiesta
    - Supportato da tutti i sistemi operativi
4. **Debugging semplice**
    - Facile vedere cosa sta succedendo
    - Log chiari degli IP testati
    - Identificazione rapida di problemi di rete
5. **Nessuna dipendenza da UDP**
    - Elimina problemi di affidabilità UDP
    - Un solo protocollo da gestire (TCP)
    - Stack networking più semplice
6. **Flessibilità**
    - Può scansionare range personalizzati
    - Supporta subnet non standard
    - Adattabile a topologie complesse

□ Svantaggi

1. **Lentezza significativa**
    - Con subnet /24: ~254 IP da testare

- Anche con parallelismo: 5-20 secondi tipici
- Timeout cumulativo può essere alto
- Esperienza utente negativa per discovery lento

2. **Carico di rete elevato**

   - Genera 254+ connessioni TCP (SYN packets)
   - Ogni tentativo = handshake TCP parziale
   - Sovraccarico su switch e router
   - Può saturare tabelle di connessione NAT

3. **Rilevamento sicurezza**

   - **IDS/IPS possono segnalare come port scan**
   - Antivirus potrebbero bloccare
   - Log di sicurezza pieni di tentativi connessione
   - Possibile blocco temporaneo IP da fail2ban

4. **Scalabilità problematica**

   - Subnet /16 (65k host): impraticabile
   - Tempo cresce linearmente con dimensione rete
   - Consumo risorse (thread/memoria) elevato
   - Difficile parallelizzare oltre certi limiti

5. **Falsi positivi**

   - Altri servizi sulla stessa porta possono rispondere
   - Necessita verifica aggiuntiva (handshake applicativo)
   - Confusione con servizi non correlati

6. **Problemi con subnet variabili**

   - Assume sempre /24 (limite arbitrario)
   - Reti con /16 o /8 non gestibili
   - Subnet /25 o /26 funzionano ma rare
   - Calcolo automatico subnet mask complesso

7. **Timeout management critico**

   - Timeout troppo breve: miss del server
   - Timeout troppo lungo: scansione lentissima
   - Condizioni di rete variabili difficili da gestire
   - Necessità di tuning per ogni ambiente

Quando Usare Scansione Subnet

 Ideale per:

- Ambienti con firewall restrittivi
- Reti Wi-Fi guest o isolate
- Quando broadcast non funziona
- Subnet molto piccole (< 50 host)
- Necessità di affidabilità assoluta
- Scopo didattico: insegnare port scanning e threading

Evitare quando:

- Reti grandi (> 254 host)
- Velocità è critica
- Policy di sicurezza rigide (IDS attivi)
- Risorse limitate (embedded systems)
- Discovery frequenti necessari

Ottimizzazioni Avanzate

Scansione Intelligente con Priorità

```
def smart_scan(self):
    # Prova prima gli IP più probabili
    likely_ips = [
        f"{base}.1",     # Gateway/server comuni
        f"{base}.2",
        f"{base}.10",
        f"{base}.100",
        f"{base}.254"
    ]

    for ip in likely_ips:
        if self.check_server(ip):
            return ip

    # Poi scansione completa
    return self.full_scan()
```

Scansione con Backoff Esponenziale

```
def adaptive_scan(self):
    batch_size = 10
    while hosts_remaining:
        batch = get_next_batch(batch_size)
        results = scan_batch(batch, timeout=current_timeout)

        if successful_rate > 0.8:
            timeout *= 0.9  # Riduci timeout
            batch_size *= 2  # Aumenta parallelismo
        else:
            timeout *= 1.2  # Aumenta timeout
            batch_size //= 2  # Riduci parallelismo
```

Cache dei Risultati

```
import json
from datetime import datetime, timedelta

def cache_server_location(self, ip):
    cache = {
        'ip': ip,
        'timestamp': datetime.now().isoformat(),
        'ttl': 3600  # 1 ora
    }
    with open('server_cache.json', 'w') as f:
        json.dump(cache, f)

def try_cached_server(self):
    try:
        with open('server_cache.json', 'r') as f:
            cache = json.load(f)

        cached_time = datetime.fromisoformat(cache['timestamp'])
        if datetime.now() - cached_time < timedelta(seconds=cache['ttl']):
            if self.check_server(cache['ip'], timeout=1):
                return cache['ip']
    except:
        pass
    return None
```

Scansione Multi-Port

```
def scan_multiple_ports(self, ip, ports=[5000, 5001, 5002]):
    for port in ports:
        if self.check_server(ip, port):
            return (ip, port)
    return None
```

Soluzione 3: File di Configurazione Condiviso

Concetto

Server e client accedono a un file condiviso (su rete o cloud) dove il server scrive il proprio IP all'avvio e il client lo legge.

Implementazione

Server - Scrittura Configurazione

```
import json
from datetime import datetime

def register_server(self):
    config = {
        'server_ip': self.get_local_ip(),
        'server_port': self.chat_port,
        'hostname': socket.gethostname(),
        'last_update': datetime.now().isoformat(),
        'status': 'online'
    }

    # Opzione 1: File di rete
    with open('//shared-drive/config/server.json', 'w') as f:
        json.dump(config, f)

    # Opzione 2: Cloud storage (es. Google Drive API)
    # upload_to_drive('server.json', config)

    # Opzione 3: Database semplice
    # save_to_sqlite('servers.db', config)
```

Client - Lettura Configurazione

```
def discover_server_from_config(self):
    try:
        with open('//shared-drive/config/server.json', 'r') as f:
            config = json.load(f)

        # Verifica se il file non è troppo vecchio
        last_update = datetime.fromisoformat(config['last_update'])
        if datetime.now() - last_update < timedelta(minutes=5):
            return config['server_ip'], config['server_port']
    except:
        return None, None
```

Pro e Contro

□ Vantaggi

1. **Semplicità estrema**
    - Implementazione banale (poche righe)
    - Nessun networking complesso
    - Debugging immediato (leggi il file)
2. **Affidabilità**
    - Non dipende da configurazioni di rete
    - Funziona attraverso subnet diverse
    - Nessun problema con firewall
3. **Persistenza**
    - Informazione sopravvive a restart
    - Storico disponibile
    - Audit trail automatico
4. **Flessibilità**
    - Può contenere metadati aggiuntivi
    - Supporto multipli server facilmente
    - Configurazione avanzata possibile
5. **Nessun traffico di rete**
    - Zero overhead di discovery
    - Non impatta performance di rete

- Scalabile a qualsiasi dimensione

□ Svantaggi

1. **Dipendenza esterna critica**
   - Richiede risorsa condivisa sempre accessibile
   - Single point of failure
   - Se condivisione non disponibile, sistema bloccato
2. **Problemi di sincronizzazione**
   - Race conditions possibili
   - Stale data (informazioni obsolete)
   - Necessita meccanismi di lock
3. **Sicurezza**
   - File accessibile da chiunque (rischio manomissione)
   - Necessita permessi condivisione corretti
   - Possibili information disclosure
4. **Performance**
   - Latenza di I/O file system
   - Overhead di rete per accesso condivisione
   - Possibili colli di bottiglia con molti client
5. **Setup complesso**
   - Richiede configurazione infrastruttura
   - Mappatura drive/mount necessaria
   - Problemi di portabilità tra OS
6. **Non scalabile enterprise**
   - Non adatto a deployment grandi
   - Gestione complicata in cloud
   - Problemi con containerizzazione

Quando Usare File Condiviso

□ **Ideale per:**

- Ambienti con NAS o file server esistente
- Reti enterprise con Active Directory
- Sviluppo/test interno
- Pochi client (<10)
- Prototipazione rapida

 Evitare quando:

- Produzione senza infrastruttura condivisa
- Deployment cloud-native
- Alta disponibilità richiesta
- Molti client concorrenti

Varianti

Database Centralizzato

```
import sqlite3

def register_to_db(self):
    conn = sqlite3.connect('//server/db/services.db')
    cursor = conn.cursor()
    cursor.execute('''
        INSERT OR REPLACE INTO servers
        (service_name, ip, port, last_seen)
        VALUES (?, ?, ?, ?)
    ''', ('chat_server', self.ip, self.port, datetime.now()))
    conn.commit()
    conn.close()
```

Redis/Key-Value Store

```
import redis

def register_to_redis(self):
    r = redis.Redis(host='shared-server', port=6379)
    r.setex('chat_server:location',
            300,  # TTL 5 minuti
            f"{self.ip}:{self.port}")
```

Soluzione 4: mDNS/Zeroconf

Concetto

Protocollo standard (RFC 6762) per service discovery senza configurazione. Usato da Bonjour (Apple), Avahi (Linux), e simili.

Implementazione con Zeroconf

Server

```
from zeroconf import Zeroconf, ServiceInfo
import socket

def register_mdns_service(self):
    zeroconf = Zeroconf()

    service_info = ServiceInfo(
        "_chatserver._tcp.local.",  # Tipo servizio
        "MyChatServer._chatserver._tcp.local.",  # Nome istanza
        addresses=[socket.inet_aton(self.get_local_ip())],
        port=self.chat_port,
        properties={
            'version': '1.0',
            'max_users': '10'
        }
    )

    zeroconf.register_service(service_info)
    return zeroconf
```

Client

```
from zeroconf import Zeroconf, ServiceBrowser

class ChatServerListener:
    def add_service(self, zeroconf, type, name):
        info = zeroconf.get_service_info(type, name)
        if info:
            address = socket.inet_ntoa(info.addresses[0])
            port = info.port
            print(f"Server trovato: {address}:{port}")
            self.server_found(address, port)

def discover_mdns_server(self):
    zeroconf = Zeroconf()
    listener = ChatServerListener()
    browser = ServiceBrowser(zeroconf, "_chatserver._tcp.local.", listener)

    # Attende discovery...
```

Pro e Contro

□ Vantaggi

1. **Standard industriale**
    - Protocollo RFC ufficiale
    - Supporto nativo in molti OS
    - Ampiamente testato e affidabile
2. **Zero configuration**
    - Nessun setup richiesto
    - Plug and play reale
    - User-friendly
3. **Feature-rich**
    - Metadati servizio inclusi
    - Supporto multipli servizi
    - TXT records per info aggiuntive
    - Service browsing
4. **Efficienza**
    - Usa multicast intelligente
    - Cache distribuita

- Minimo traffico di rete

5. **Interoperabilità**

   - Funziona con servizi esistenti
   - Integrazione con Bonjour/Avahi
   - Standard cross-platform

□ Svantaggi

1. **Dipendenze esterne**

   - Richiede libreria `zeroconf` o `python-zeroconf`
   - Installazione dipendenze su ogni client
   - Potenziali problemi di versioning

2. **Complessità**

   - Curva di apprendimento ripida
   - Documentazione dispersa
   - Debugging complicato

3. **Limitazioni di rete**

   - Non attraversa router (come broadcast)
   - Problemi con alcune configurazioni firewall
   - Richiede multicast funzionante

4. **Overhead**

   - Libreria pesante per uso semplice
   - Consumo memoria maggiore
   - Startup time più lungo

5. **Non didattico**

   - Black box per studenti
   - Nasconde dettagli implementativi
   - Poco formativo su networking

Quando Usare mDNS

□ **Ideale per:**

- Applicazioni professionali
- Prodotti commerciali
- Integrazione con ecosistema Apple/Linux
- Necessità di standard riconosciuto
- Feature avanzate richieste

□ **Evitare quando:**

- Scopo didattico (troppo astratto)
- Dipendenze non accettabili
- Deployment semplificato necessario
- Controllo completo richiesto

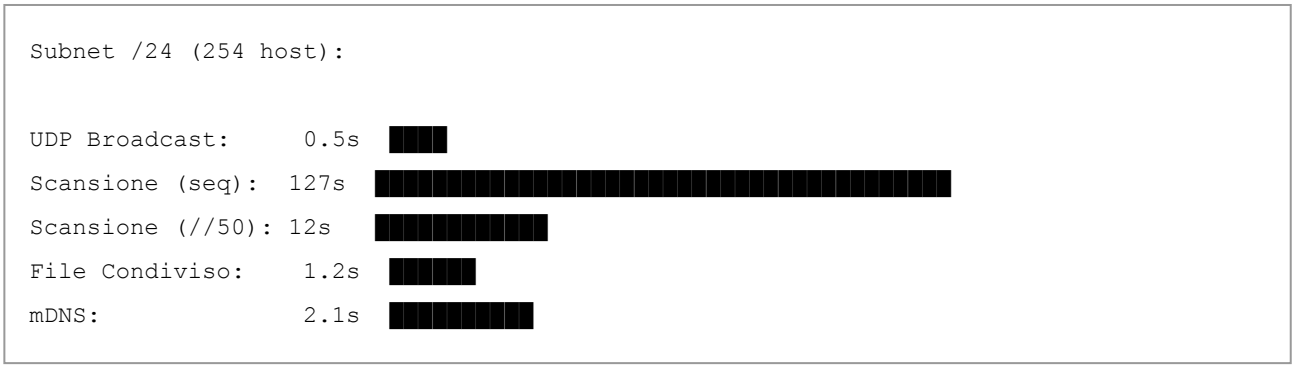
Confronto Comparativo

Tabella Comparativa Completa

Criterio	UDP Broadcast	Scansione Subnet	File Condiviso	mDNS/Zeroconf
Velocità	□□□□□ (<1s)	□□ (5-20s)	□□□□ (1-2s)	□□□□ (1-3s)
Affidabilità	□□□	□□□□□	□□□	□□□□
Semplicità Codice	□□□□	□□□	□□□□□	□□
Efficienza Rete	□□□□□	□□	□□□□□	□□□□
Compatibilità Firewall	□□	□□□□□	□□□□□	□□□
Scalabilità	□□□□□	□□	□□□	□□□□
Zero Config	□□□□□	□□□□□	□□	□□□□□
Valore Didattico	□□□□□	□□□□	□□	□□
Prod-Ready	□□□	□□□	□□	□□□□□
Setup Richiesto	Nessuno	Nessuno	Medio	Librerie
Dipendenze	Nessuna	Nessuna	Infrastruttura	python-zeroconf
Cross-Subnet	□ No	□ No	□ Sì	□ No
Sicurezza IDS	□ Ok	⚠ Port scan	□ Ok	□ Ok
Costo CPU	Basso	Alto	Basso	Medio
Costo Memoria	Basso	Medio	Basso	Alto

Metriche Prestazionali

Tempo di Discovery (Media)



Traffico di Rete Generato

```
UDP Broadcast:    ~500 bytes   ▌
Scansione /24:    ~15 KB       ████████████████
File Condiviso:   0 bytes      (I/O file)
mDNS:             ~2 KB        ███
```

False Positive Rate

```
UDP Broadcast:    < 1%   (protocollo custom)
Scansione:        5-10%  (altri servizi su stessa porta)
File Condiviso:   < 1%   (lettura diretta)
mDNS:             < 1%   (service type specifico)
```

Matrice Decisionale

Caratteristiche Ambiente → Soluzione Raccomandata

Rete locale piccola Firewall permissivi Scopo didattico	UDP Broadcast □
Firewall restrittivi Wi-Fi guest Broadcast non funziona	Scansione Subnet (solo se < 50 host)
NAS/file server esiste Ambiente corporate Pochi client statici	File Condiviso (prototipi e test)
Prodotto commerciale Standard richiesto Integrazione Bonjour	mDNS/Zeroconf □

Raccomandazioni per Contesto Didattico

Per Corsi di Sistemi e Reti

Approccio Pedagogico Progressivo

Lezione 1: Concetti Base

- Problema: hardcoded IP addresses
- Introduzione al service discovery
- Soluzione manuale (file configurazione)

Lezione 2: UDP Broadcast □ RACCOMANDATO

- Differenza UDP vs TCP
- Concetto di broadcast
- Implementazione pratica
- Debug con Wireshark

Lezione 3: Scansione e Threading

- Port scanning etico
- Concurrency in Python
- ThreadPoolExecutor
- Ottimizzazioni performance

Lezione 4: Protocolli Standard

- Introduzione mDNS
- Analisi RFC
- Confronto soluzioni custom vs standard

Lab Pratico Suggerito

Esercitazione Completa (4 ore)

Parte 1: Implementazione Base (90 min)

1. Implementare chat server/client basico con IP hardcoded
2. Identificare limiti e problemi
3. Discussione: quali soluzioni esistono?

Parte 2: UDP Broadcast (90 min)

1. Aggiungere discovery service al server
2. Implementare discovery sul client
3. Testing in rete locale

4. Cattura traffico con Wireshark
5. Analisi pacchetti UDP

Parte 3: Problematiche Reali (45 min)

1. Simulare firewall: bloccare UDP broadcast
2. Tentare discovery → fallisce
3. Implementare fallback a scansione subnet
4. Confrontare prestazioni

Parte 4: Discussione e Ottimizzazioni (45 min)

1. Analisi pro/contro di ogni soluzione
2. Quando usare quale approccio
3. Sicurezza: autenticazione discovery
4. Progetti avanzati: load balancing, failover

Progetti Studente Avanzati

Livello Intermedio

- **Hybrid Discovery**: prova broadcast, poi scansione come fallback
- **Multi-Server Load Balancer**: client sceglie server meno carico
- **Server Heartbeat**: verifica periodica che server sia ancora attivo
- **Encrypted Discovery**: autenticazione tramite challenge-response

Livello Avanzato

- **Service Registry**: server centrale che traccia tutti i servizi
- **Geo-Discovery**: preferenza server geograficamente vicini
- **Fault Tolerance**: failover automatico se server primario cade
- **Cross-Subnet Discovery**: relay nodes per attraversare router

Valutazione e Criteri

Rubrica Valutazione Progetto

Criterio	Punti	Descrizione
Funzionalità	30	Discovery funziona in condizioni normali
Robustezza	20	Gestione errori e timeout
Performance	15	Velocità discovery accettabile
Codice	15	Leggibilità, commenti, struttura
Documentazione	10	README, diagrammi, esempi
Testing	10	Test cases e validazione

Domande di Verifica Comprensione

1. **Concettuali**
    - Perché UDP per discovery e TCP per chat?
    - Cosa succede se due server rispondono al broadcast?
    - Quali problemi causa il broadcast in reti grandi?
2. **Pratiche**
    - Come modificheresti il codice per supportare IPv6?
    - Come implementeresti retry con backoff esponenziale?
    - Come garantiresti che solo server autorizzati rispondano?
3. **Analisi**
    - Confronta overhead di broadcast vs scansione in subnet /16
    - Stima il tempo di scansione con 1000 host e timeout 0.5s
    - Quale soluzione useresti per IoT con 100+ dispositivi?

Risorse Didattiche Aggiuntive

Tools Consigliati

- **Wireshark**: analisi traffico di rete (broadcast, multicast)
- **nmap**: studio port scanning professionale
- **netcat**: testing manuale connessioni TCP/UDP
- **iperf**: misurazione performance di rete

Approfondimenti

- RFC 6762 (mDNS): lettura standard ufficiale
- Stevens "Unix Network Programming": capitoli su broadcast/multicast
- Beej's Guide to Network Programming: reference pratica
- Python `socket` documentation: API dettagliata

Progetti Open Source da Studiare

- **Avahi**: implementazione mDNS Linux
- **Bonjour**: implementazione Apple
- **Syncthing**: discovery P2P avanzato
- **BitTorrent DHT**: distributed hash table per discovery

Conclusioni e Best Practices

Decision Tree Finale

```
Devo implementare service discovery?
│
├─ Scopo didattico?
│  ├─ Sì → UDP Broadcast (insegna networking)
│  └─ No ↓
│
├─ Ambiente controllato (lab/ufficio)?
│  ├─ Sì → UDP Broadcast (veloce e semplice)
│  └─ No ↓
│
├─ Firewall bloccano broadcast?
│  ├─ Sì → Scansione Subnet (se < 100 host)
│  └─ No → UDP Broadcast
│
├─ Necessario attraversare router?
│  ├─ Sì → File Condiviso o Server Registry
│  └─ No → UDP Broadcast
│
├─ Prodotto commerciale?
│  └─ Sì → mDNS/Zeroconf (standard)
│
└─ Prototipo rapido interno?
   └─ Sì → File Condiviso (semplicissimo)
```

Checklist Implementazione

Prima di Iniziare

- [ ] Definire requisiti: velocità vs affidabilità
- [ ] Analizzare ambiente di rete target
- [ ] Verificare policy firewall
- [ ] Decidere se cross-subnet necessario
- [ ] Valutare competenze team

Durante Sviluppo

- [ ] Implementare timeout appropriati
- [ ] Gestire eccezioni di rete
- [ ] Aggiungere logging dettagliato
- [ ] Testare con Wireshark
- [ ] Validare su reti diverse

Prima del Deploy

- [ ] Test con firewall attivo
- [ ] Test con antivirus attivo
- [ ] Misurare performance reali
- [ ] Documentare requisiti di rete
- [ ] Preparare troubleshooting guide

Errori Comuni da Evitare

1. **Timeout troppo brevi**: miss dei server su reti lente
2. **Nessun fallback**: discovery fallisce, app inutilizzabile
3. **Mancanza logging**: debugging impossibile
4. **Ignorare IPv6**: sempre più comune
5. **Hardcoded subnet /24**: reti diverse usano /16, /22, ecc.
6. **Nessuna autenticazione**: rischi di sicurezza
7. **Broadcasting eccessivo**: congestione di rete
8. **Race conditions**: multipli server, nessuna sincronizzazione

Pattern Avanzati

Hybrid Approach (Raccomandato per Produzione)

```
def discover_server_robust(self):
    # 1. Prova cache locale
    if server := self.try_cache():
        return server

    # 2. Prova UDP broadcast (veloce)
    if server := self.udp_discovery(timeout=2):
        self.cache_server(server)
        return server

    # 3. Fallback a scansione (affidabile)
    if server := self.subnet_scan(max_workers=30):
        self.cache_server(server)
        return server

    # 4. Fallback manuale
    return self.manual_input()
```

Service Registry Pattern

```
# Server centrale mantiene lista servizi
class ServiceRegistry:
    def register(self, service_name, ip, port, ttl=300):
        self.services[service_name] = {
            'ip': ip,
            'port': port,
            'expires': time.time() + ttl
        }

    def discover(self, service_name):
        if service_name in self.services:
            if time.time() < self.services[service_name]['expires']:
                return self.services[service_name]
        return None
```

Sicurezza: Considerazioni Critiche

Autenticazione Discovery

```
import hmac
import hashlib

SECRET_KEY = b'shared-secret-key'

# Server
def authenticate_response(self, message):
    signature = hmac.new(SECRET_KEY, message.encode(), hashlib.sha256).hexdigest()
    return f"{message}|{signature}"

# Client
def verify_response(self, response):
    message, signature = response.rsplit('|', 1)
    expected = hmac.new(SECRET_KEY, message.encode(), hashlib.sha256).hexdigest()
    return hmac.compare_digest(signature, expected)
```

Rate Limiting

```
from collections import defaultdict
import time

class RateLimiter:
    def __init__(self, max_requests=5, window=60):
        self.requests = defaultdict(list)
        self.max_requests = max_requests
        self.window = window

    def allow_request(self, ip):
        now = time.time()
        # Rimuovi richieste vecchie
        self.requests[ip] = [t for t in self.requests[ip]
                             if now - t < self.window]

        if len(self.requests[ip]) < self.max_requests:
            self.requests[ip].append(now)
            return True
        return False
```

Riferimenti e Risorse

Standard e RFC

- **RFC 6762**: Multicast DNS
- **RFC 2782**: DNS SRV Records
- **RFC 919**: Broadcasting Internet Datagrams

Libri Consigliati

- "Unix Network Programming" - W. Richard Stevens
- "Computer Networks" - Andrew S. Tanenborough
- "Effective Python" - Brett Slatkin (per threading)

Documentazione Online

- Python `socket` module: https://docs.python.org/3/library/socket.html (https://docs.python.org/3/library/socket.html)
- Python `ipaddress` module: https://docs.python.org/3/library/ipaddress.html (https://docs.python.org/3/library/ipaddress.html)
- Zeroconf documentation: https://python-zeroconf.readthedocs.io/ (https://python-zeroconf.readthedocs.io/)

Tools e Software

- **Wireshark**: https://www.wireshark.org/ (https://www.wireshark.org/)
- **nmap**: https://nmap.org/ (https://nmap.org/)
- **Avahi**: https://www.avahi.org/ (https://www.avahi.org/)

Appendice: Codice Completo Implementazioni

A. UDP Broadcast Discovery (Versione Estesa)

```
# Vedi artifact chat_server_discovery
# Vedi artifact chat_client_discovery
```

B. Scansione Subnet (Versione Estesa)

```
# Vedi artifact chat_client_subnet_scan
```

C. Esempio File Condiviso

```
# server_config_writer.py
import json
import socket
from datetime import datetime

class ConfigFileServer:
    def __init__(self, config_path='\\\\shared\\server_config.json'):
        self.config_path = config_path
        self.port = 5000

    def register(self):
        config = {
            'ip': self.get_local_ip(),
            'port': self.port,
            'hostname': socket.gethostname(),
            'timestamp': datetime.now().isoformat(),
            'status': 'active'
        }

        try:
            with open(self.config_path, 'w') as f:
                json.dump(config, f, indent=2)
            print(f"Server registrato: {config}")
        except Exception as e:
            print(f"Errore registrazione: {e}")

    def get_local_ip(self):
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.connect(("8.8.8.8", 80))
        ip = s.getsockname()[0]
        s.close()
        return ip

# client_config_reader.py
import json
from datetime import datetime, timedelta

class ConfigFileClient:
    def __init__(self, config_path='\\\\shared\\server_config.json'):
        self.config_path = config_path

    def discover_server(self, max_age_minutes=5):
        try:
```

```
            with open(self.config_path, 'r') as f:
                config = json.load(f)

            # Verifica che il config non sia troppo vecchio
            timestamp = datetime.fromisoformat(config['timestamp'])
            age = datetime.now() - timestamp

            if age < timedelta(minutes=max_age_minutes):
                print(f"Server trovato: {config['ip']}:{config['port']}")
                return config['ip'], config['port']
            else:
                print(f"Config obsoleto (età: {age})")
                return None, None
        except FileNotFoundError:
            print("File di configurazione non trovato")
            return None, None
        except Exception as e:
            print(f"Errore lettura config: {e}")
            return None, None
```

D. Esempio mDNS/Zeroconf Base

```python
# server_mdns.py
from zeroconf import Zeroconf, ServiceInfo
import socket
import time

class MDNSServer:
    def __init__(self, port=5000):
        self.port = port
        self.zeroconf = None
        self.service_info = None

    def register_service(self):
        self.zeroconf = Zeroconf()

        # Definizione servizio
        service_type = "_chatserver._tcp.local."
        service_name = f"MyChatServer.{service_type}"

        # Ottieni IP locale
        local_ip = self.get_local_ip()

        # Crea service info
        self.service_info = ServiceInfo(
            service_type,
            service_name,
            addresses=[socket.inet_aton(local_ip)],
            port=self.port,
            properties={
                b'version': b'1.0',
                b'description': b'Python Chat Server'
            },
            server=f"{socket.gethostname()}.local."
        )

        # Registra
        self.zeroconf.register_service(self.service_info)
        print(f"Servizio mDNS registrato: {service_name}")
        print(f"IP: {local_ip}, Porta: {self.port}")

    def unregister_service(self):
        if self.zeroconf and self.service_info:
            self.zeroconf.unregister_service(self.service_info)
            self.zeroconf.close()
```

```
            print("Servizio mDNS deregistrato")

    def get_local_ip(self):
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.connect(("8.8.8.8", 80))
        ip = s.getsockname()[0]
        s.close()
        return ip

# client_mdns.py
from zeroconf import Zeroconf, ServiceBrowser, ServiceStateChange
import socket
import time

class MDNSListener:
    def __init__(self):
        self.server_found = False
        self.server_ip = None
        self.server_port = None

    def on_service_state_change(self, zeroconf, service_type, name, state_change):
        if state_change is ServiceStateChange.Added:
            info = zeroconf.get_service_info(service_type, name)
            if info:
                self.server_ip = socket.inet_ntoa(info.addresses[0])
                self.server_port = info.port
                self.server_found = True
                print(f"Server trovato: {self.server_ip}:{self.server_port}")

class MDNSClient:
    def discover_server(self, timeout=5):
        zeroconf = Zeroconf()
        listener = MDNSListener()

        browser = ServiceBrowser(
            zeroconf,
            "_chatserver._tcp.local.",
            handlers=[listener.on_service_state_change]
        )

        # Attendi discovery
        start_time = time.time()
        while not listener.server_found and (time.time() - start_time) < timeout:
            time.sleep(0.1)
```

```
    zeroconf.close()

    if listener.server_found:
        return listener.server_ip, listener.server_port
    return None, None
```

Glossario Tecnico

- **Broadcast**: Invio di un messaggio a tutti i dispositivi in una subnet
- **Multicast**: Invio a un gruppo specifico di destinatari
- **Unicast**: Invio point-to-point a un singolo destinatario
- **Service Discovery**: Meccanismo per trovare servizi in rete automaticamente
- **mDNS**: Multicast DNS, protocollo per discovery locale
- **Subnet**: Sottorete, divisione logica di una rete IP
- **CIDR**: Classless Inter-Domain Routing, notazione per subnet (es. /24)
- **Port Scanning**: Tecnica per identificare porte aperte su un host
- **TTL**: Time To Live, durata di validità di un dato
- **Zeroconf**: Zero Configuration Networking, set di tecnologie per auto-config

---

Documento versione 1.0

Ultima modifica: Gennaio 2026

Autore: Sistema di documentazione tecnica

Guida Completa: Service Discovery in Reti Locali

Indice

Introduzione al Problema

Il Problema

In un'applicazione client-server tradizionale, il client deve conoscere a priori l'indirizzo IP del server. Questo approccio presenta diversi limiti:

- **Rigidità**: ogni volta che il server cambia macchina, bisogna modificare il codice client
- **Manutenzione**: in ambienti con più server, gestire gli IP diventa complesso
- **Scalabilità**: difficile distribuire l'applicazione su diverse reti
- **Usabilità**: l'utente finale deve conoscere dettagli tecnici (IP, porta)

La Soluzione: Service Discovery

Il **Service Discovery** è un meccanismo che permette ai client di trovare automaticamente i servizi disponibili sulla rete senza configurazione manuale.

Requisiti base:

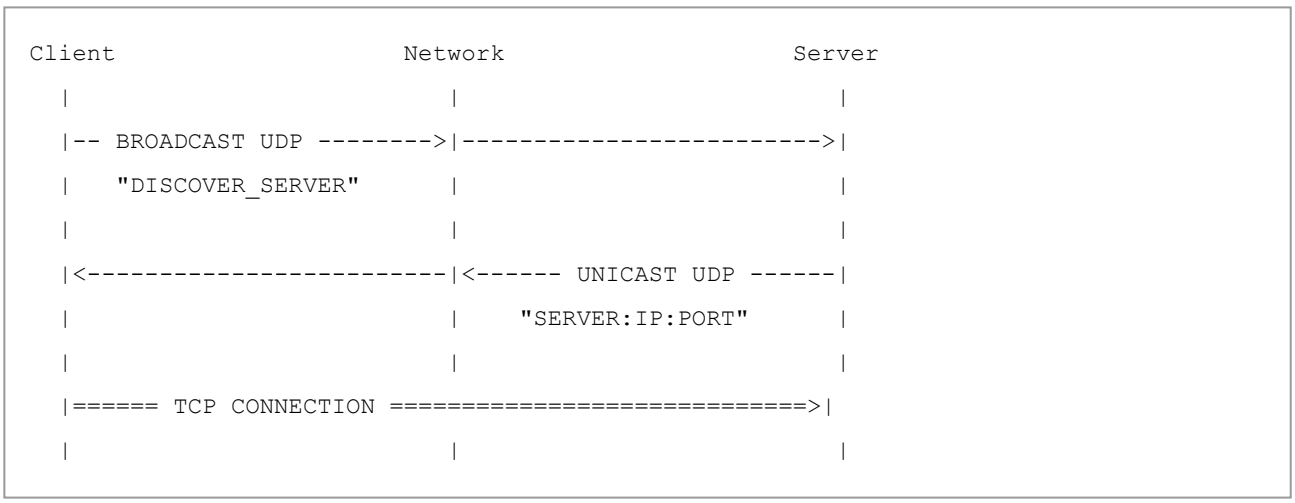
- Automatico (nessun intervento umano)
- Affidabile (deve trovare il server se esiste)
- Veloce (tempi di discovery ragionevoli)
- Semplice da implementare

Soluzione 1: UDP Broadcast Discovery

Concetto

Il client invia un messaggio **broadcast** UDP sulla rete locale. Tutti i dispositivi ricevono il messaggio, ma solo il server risponde identificandosi.

Architettura



Implementazione Dettagliata

Server - Servizio Discovery

```
def discovery_service(self):
    # Socket UDP per ricevere broadcast
    discovery_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    discovery_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    # Bind su TUTTE le interfacce di rete
    discovery_socket.bind(('', self.discovery_port))

    while self.running:
        data, addr = discovery_socket.recvfrom(1024)
        message = data.decode('utf-8')

        if message == "DISCOVER_CHAT_SERVER":
            # Risponde con IP e porta del servizio chat
            response = f"CHAT_SERVER:{self.get_local_ip()}:{self.chat_port}"
            discovery_socket.sendto(response.encode('utf-8'), addr)
```

Punti chiave:

- `SO_REUSEADDR`: permette di riutilizzare la porta anche se non completamente chiusa
- `bind(('', port))`: ascolta su tutte le interfacce di rete (0.0.0.0)
- Risposta unicast: torna solo al client richiedente

Client - Discovery

```
def discover_server(self, timeout=5):
    discovery_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # Abilita il broadcast
    discovery_socket.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
    discovery_socket.settimeout(timeout)

    # Invia broadcast a TUTTI i dispositivi nella subnet
    discovery_socket.sendto(
        "DISCOVER_CHAT_SERVER".encode('utf-8'),
        ('<broadcast>', self.discovery_port)
    )

    # Attende risposta
    data, addr = discovery_socket.recvfrom(1024)
    # Parsing della risposta...
```

Punti chiave:

- `SO_BROADCAST`: necessario per inviare pacchetti broadcast
- `<broadcast>`: indirizzo speciale 255.255.255.255
- `settimeout()`: evita attese infinite se nessun server risponde

Pro e Contro

□ Vantaggi

1. **Velocità eccezionale**
    - Discovery quasi istantaneo (< 1 secondo)
    - Un solo pacchetto broadcast necessario
    - Nessuna iterazione su IP multipli
2. **Efficienza di rete**
    - Traffico minimo generato
    - Un solo pacchetto broadcast + una risposta unicast
    - Non sovraccarica la rete
3. **Scalabilità**
    - Funziona con qualsiasi numero di dispositivi
    - Costo computazionale costante O(1)
    - Non dipende dalla dimensione della subnet
4. **Semplicità concettuale**

- Logica diretta e intuitiva
- Facile debugging con Wireshark
- Pattern standard nell'industria

5. **Supporto multipli server**

   - Più server possono rispondere contemporaneamente
   - Il client può scegliere o usare tutti i server trovati
   - Utile per load balancing o fault tolerance

6. **Didattico**

   - Insegna la differenza tra UDP e TCP
   - Mostra l'uso del broadcast
   - Introduce concetti di service discovery

□ Svantaggi

1. **Limitazioni di rete**

   - **Non attraversa i router**: funziona solo nella subnet locale (broadcast domain)
   - **Segmentazione VLAN**: non funziona tra VLAN diverse
   - **VPN**: problemi con alcune configurazioni VPN

2. **Firewall e sicurezza**

   - Molti firewall bloccano broadcast UDP per default
   - Windows Firewall può richiedere regole esplicite
   - Alcuni antivirus segnalano come sospetto

3. **Affidabilità UDP**

   - UDP non garantisce consegna (unreliable protocol)
   - Pacchetti possono essere persi senza notifica
   - Nessun acknowledgment o retry automatico

4. **Potenziali collisioni**

   - Se più server rispondono simultaneamente, possibili conflitti
   - Necessita logica di gestione risposte multiple
   - Possibile confusione con servizi sulla stessa porta

5. **Congestione broadcast**

   - In reti molto grandi, il broadcast aggiunge traffico
   - Ogni dispositivo deve processare il pacchetto
   - Switch possono limitare broadcast rate

6. **Problemi con Wi-Fi**

- Access Point possono filtrare broadcast
- Reti guest spesso bloccano broadcast tra client
- Modalità risparmio energetico può perdere pacchetti

Quando Usare UDP Broadcast

✅ Ideale per:

- Reti locali piccole/medie (< 254 host)
- Ambienti controllati (lab, ufficio)
- Applicazioni dove velocità è prioritaria
- Prototipi e proof of concept
- Contesto didattico (insegna networking)

❌ Evitare quando:

- Attraversamento router necessario
- Reti enterprise con policy rigide
- Ambienti con firewall restrittivi
- Necessità di garanzie di consegna

Varianti e Ottimizzazioni

Multicast invece di Broadcast

```
# Server
MCAST_GRP = '224.1.1.1'
sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP,
                socket.inet_aton(MCAST_GRP) + socket.inet_aton('0.0.0.0'))

# Client
sock.sendto(message, (MCAST_GRP, port))
```

Vantaggi del Multicast:

- Più efficiente del broadcast (solo interessati ricevono)
- Può attraversare router se configurato (con IGMP)
- Minore impatto sulla rete

Svantaggi:

- Configurazione più complessa
- Non sempre supportato da router consumer
- Richiede setup IGMP

Retry e Timeout Adattivo

```
def discover_server_with_retry(self, max_attempts=3):
    for attempt in range(max_attempts):
        timeout = 2 * (attempt + 1)  # Timeout crescente
        if self.discover_server(timeout):
            return True
    return False
```

Autenticazione e Sicurezza

```
import hmac
import hashlib

# Server
def verify_request(self, data, shared_secret):
    message, signature = data.split('|')
    expected = hmac.new(shared_secret, message.encode(), hashlib.sha256).hexdigest()
    return hmac.compare_digest(signature, expected)
```

Soluzione 2: Scansione Diretta della Subnet

Concetto

Il client determina la subnet locale (es. 192.168.1.0/24) e tenta di connettersi sequenzialmente o in parallelo a ogni IP sulla porta del server.

Architettura

```
Client                    Network
  |
  |-- Calcola subnet: 192.168.1.0/24
  |
  |-- TCP SYN --> 192.168.1.1:5000  (timeout/refused)
  |-- TCP SYN --> 192.168.1.2:5000  (timeout/refused)
  |-- TCP SYN --> 192.168.1.3:5000  (SUCCESS!) ✓
  |
  |====== Connessione stabilita ======>
```

Implementazione Dettagliata

Determinazione Subnet

```
def get_local_network(self):
    # Ottiene IP locale
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect(("8.8.8.8", 80))  # Non invia realmente dati
    local_ip = s.getsockname()[0]
    s.close()

    # Crea oggetto rete con subnet /24
    network = ipaddress.IPv4Network(f"{local_ip}/24", strict=False)
    # Es: 192.168.1.100 -> 192.168.1.0/24

    return network
```

Punti chiave:

- Trucco con `connect()` a 8.8.8.8: determina quale interfaccia userebbe il sistema
- `ipaddress.IPv4Network`: modulo standard Python per gestione IP
- `strict=False`: accetta IP host e calcola network address
- `/24`: subnet mask 255.255.255.0 (254 host utilizzabili)

Scansione con Threading

```
def scan_network(self, max_workers=50):
    network = self.get_local_network()
    found_servers = []

    with concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as executor:
        # Crea un future per ogni IP
        future_to_ip = {
            executor.submit(self.check_server, ip): ip
            for ip in network.hosts()
        }

        # Processa risultati man mano che arrivano
        for future in concurrent.futures.as_completed(future_to_ip):
            ip = future_to_ip[future]
            if future.result():
                found_servers.append(str(ip))

    return found_servers
```

Punti chiave:

- `ThreadPoolExecutor`: pool di thread per esecuzione parallela
- `max_workers=50`: numero massimo thread simultanei (bilanciamento carico/velocità)
- `as_completed()`: processa risultati appena disponibili (non attende tutti)
- `network.hosts()`: esclude network address e broadcast (x.x.x.0 e x.x.x.255)

Verifica Server

```
def check_server(self, ip, timeout=0.5):
    try:
        test_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        test_socket.settimeout(timeout)

        # connect_ex() ritorna 0 se successo, errno altrimenti
        result = test_socket.connect_ex((str(ip), self.server_port))

        test_socket.close()
        return result == 0  # True se connessione riuscita
    except:
        return False
```

Punti chiave:

- `connect_ex()`: versione non-blocking di connect() che ritorna errno
- Timeout breve (0.5s): bilancia velocità e affidabilità
- Gestione eccezioni: host irraggiungibili possono causare exception

Pro e Contro

□ Vantaggi

1. **Affidabilità massima**
    - Funziona sempre se il server è raggiungibile via TCP
    - Non dipende da broadcast o multicast
    - Nessuna dipendenza da configurazioni di rete speciali
2. **Attraversamento ostacoli**
    - Funziona con firewall che bloccano broadcast
    - Compatibile con reti Wi-Fi guest (isolamento client)
    - Nessun problema con policy di rete restrittive
3. **Compatibilità universale**
    - Funziona su qualsiasi rete TCP/IP
    - Nessuna configurazione speciale richiesta
    - Supportato da tutti i sistemi operativi
4. **Debugging semplice**
    - Facile vedere cosa sta succedendo
    - Log chiari degli IP testati
    - Identificazione rapida di problemi di rete
5. **Nessuna dipendenza da UDP**
    - Elimina problemi di affidabilità UDP
    - Un solo protocollo da gestire (TCP)
    - Stack networking più semplice
6. **Flessibilità**
    - Può scansionare range personalizzati
    - Supporta subnet non standard
    - Adattabile a topologie complesse

□ Svantaggi

1. **Lentezza significativa**
    - Con subnet /24: ~254 IP da testare

- Anche con parallelismo: 5-20 secondi tipici
- Timeout cumulativo può essere alto
- Esperienza utente negativa per discovery lento

2. **Carico di rete elevato**

   - Genera 254+ connessioni TCP (SYN packets)
   - Ogni tentativo = handshake TCP parziale
   - Sovraccarico su switch e router
   - Può saturare tabelle di connessione NAT

3. **Rilevamento sicurezza**

   - **IDS/IPS possono segnalare come port scan**
   - Antivirus potrebbero bloccare
   - Log di sicurezza pieni di tentativi connessione
   - Possibile blocco temporaneo IP da fail2ban

4. **Scalabilità problematica**

   - Subnet /16 (65k host): impraticabile
   - Tempo cresce linearmente con dimensione rete
   - Consumo risorse (thread/memoria) elevato
   - Difficile parallelizzare oltre certi limiti

5. **Falsi positivi**

   - Altri servizi sulla stessa porta possono rispondere
   - Necessita verifica aggiuntiva (handshake applicativo)
   - Confusione con servizi non correlati

6. **Problemi con subnet variabili**

   - Assume sempre /24 (limite arbitrario)
   - Reti con /16 o /8 non gestibili
   - Subnet /25 o /26 funzionano ma rare
   - Calcolo automatico subnet mask complesso

7. **Timeout management critico**

   - Timeout troppo breve: miss del server
   - Timeout troppo lungo: scansione lentissima
   - Condizioni di rete variabili difficili da gestire
   - Necessità di tuning per ogni ambiente

Quando Usare Scansione Subnet

 Ideale per:

- Ambienti con firewall restrittivi
- Reti Wi-Fi guest o isolate
- Quando broadcast non funziona
- Subnet molto piccole (< 50 host)
- Necessità di affidabilità assoluta
- Scopo didattico: insegnare port scanning e threading

Evitare quando:

- Reti grandi (> 254 host)
- Velocità è critica
- Policy di sicurezza rigide (IDS attivi)
- Risorse limitate (embedded systems)
- Discovery frequenti necessari

Ottimizzazioni Avanzate

Scansione Intelligente con Priorità

```
def smart_scan(self):
    # Prova prima gli IP più probabili
    likely_ips = [
        f"{base}.1",     # Gateway/server comuni
        f"{base}.2",
        f"{base}.10",
        f"{base}.100",
        f"{base}.254"
    ]

    for ip in likely_ips:
        if self.check_server(ip):
            return ip

    # Poi scansione completa
    return self.full_scan()
```

Scansione con Backoff Esponenziale

```
def adaptive_scan(self):
    batch_size = 10
    while hosts_remaining:
        batch = get_next_batch(batch_size)
        results = scan_batch(batch, timeout=current_timeout)

        if successful_rate > 0.8:
            timeout *= 0.9  # Riduci timeout
            batch_size *= 2  # Aumenta parallelismo
        else:
            timeout *= 1.2  # Aumenta timeout
            batch_size //= 2  # Riduci parallelismo
```

Cache dei Risultati

```
import json
from datetime import datetime, timedelta

def cache_server_location(self, ip):
    cache = {
        'ip': ip,
        'timestamp': datetime.now().isoformat(),
        'ttl': 3600  # 1 ora
    }
    with open('server_cache.json', 'w') as f:
        json.dump(cache, f)

def try_cached_server(self):
    try:
        with open('server_cache.json', 'r') as f:
            cache = json.load(f)

        cached_time = datetime.fromisoformat(cache['timestamp'])
        if datetime.now() - cached_time < timedelta(seconds=cache['ttl']):
            if self.check_server(cache['ip'], timeout=1):
                return cache['ip']
    except:
        pass
    return None
```

Scansione Multi-Port

```
def scan_multiple_ports(self, ip, ports=[5000, 5001, 5002]):
    for port in ports:
        if self.check_server(ip, port):
            return (ip, port)
    return None
```

Soluzione 3: File di Configurazione Condiviso

Concetto

Server e client accedono a un file condiviso (su rete o cloud) dove il server scrive il proprio IP all'avvio e il client lo legge.

Implementazione

Server - Scrittura Configurazione

```
import json
from datetime import datetime

def register_server(self):
    config = {
        'server_ip': self.get_local_ip(),
        'server_port': self.chat_port,
        'hostname': socket.gethostname(),
        'last_update': datetime.now().isoformat(),
        'status': 'online'
    }

    # Opzione 1: File di rete
    with open('//shared-drive/config/server.json', 'w') as f:
        json.dump(config, f)

    # Opzione 2: Cloud storage (es. Google Drive API)
    # upload_to_drive('server.json', config)

    # Opzione 3: Database semplice
    # save_to_sqlite('servers.db', config)
```

Client - Lettura Configurazione

```
def discover_server_from_config(self):
    try:
        with open('//shared-drive/config/server.json', 'r') as f:
            config = json.load(f)

        # Verifica se il file non è troppo vecchio
        last_update = datetime.fromisoformat(config['last_update'])
        if datetime.now() - last_update < timedelta(minutes=5):
            return config['server_ip'], config['server_port']
    except:
        return None, None
```

Pro e Contro

□ Vantaggi

1. **Semplicità estrema**
    - Implementazione banale (poche righe)
    - Nessun networking complesso
    - Debugging immediato (leggi il file)
2. **Affidabilità**
    - Non dipende da configurazioni di rete
    - Funziona attraverso subnet diverse
    - Nessun problema con firewall
3. **Persistenza**
    - Informazione sopravvive a restart
    - Storico disponibile
    - Audit trail automatico
4. **Flessibilità**
    - Può contenere metadati aggiuntivi
    - Supporto multipli server facilmente
    - Configurazione avanzata possibile
5. **Nessun traffico di rete**
    - Zero overhead di discovery
    - Non impatta performance di rete

- Scalabile a qualsiasi dimensione

□ Svantaggi

1. **Dipendenza esterna critica**
   - Richiede risorsa condivisa sempre accessibile
   - Single point of failure
   - Se condivisione non disponibile, sistema bloccato
2. **Problemi di sincronizzazione**
   - Race conditions possibili
   - Stale data (informazioni obsolete)
   - Necessita meccanismi di lock
3. **Sicurezza**
   - File accessibile da chiunque (rischio manomissione)
   - Necessita permessi condivisione corretti
   - Possibili information disclosure
4. **Performance**
   - Latenza di I/O file system
   - Overhead di rete per accesso condivisione
   - Possibili colli di bottiglia con molti client
5. **Setup complesso**
   - Richiede configurazione infrastruttura
   - Mappatura drive/mount necessaria
   - Problemi di portabilità tra OS
6. **Non scalabile enterprise**
   - Non adatto a deployment grandi
   - Gestione complicata in cloud
   - Problemi con containerizzazione

Quando Usare File Condiviso

□ **Ideale per:**

- Ambienti con NAS o file server esistente
- Reti enterprise con Active Directory
- Sviluppo/test interno
- Pochi client (<10)
- Prototipazione rapida

 Evitare quando:

- Produzione senza infrastruttura condivisa
- Deployment cloud-native
- Alta disponibilità richiesta
- Molti client concorrenti

Varianti

Database Centralizzato

```
import sqlite3

def register_to_db(self):
    conn = sqlite3.connect('//server/db/services.db')
    cursor = conn.cursor()
    cursor.execute('''
        INSERT OR REPLACE INTO servers
        (service_name, ip, port, last_seen)
        VALUES (?, ?, ?, ?)
    ''', ('chat_server', self.ip, self.port, datetime.now()))
    conn.commit()
    conn.close()
```

Redis/Key-Value Store

```
import redis

def register_to_redis(self):
    r = redis.Redis(host='shared-server', port=6379)
    r.setex('chat_server:location',
            300,  # TTL 5 minuti
            f"{self.ip}:{self.port}")
```

Soluzione 4: mDNS/Zeroconf

Concetto

Protocollo standard (RFC 6762) per service discovery senza configurazione. Usato da Bonjour (Apple), Avahi (Linux), e simili.

Implementazione con Zeroconf

Server

```
from zeroconf import Zeroconf, ServiceInfo
import socket

def register_mdns_service(self):
    zeroconf = Zeroconf()

    service_info = ServiceInfo(
        "_chatserver._tcp.local.",  # Tipo servizio
        "MyChatServer._chatserver._tcp.local.",  # Nome istanza
        addresses=[socket.inet_aton(self.get_local_ip())],
        port=self.chat_port,
        properties={
            'version': '1.0',
            'max_users': '10'
        }
    )

    zeroconf.register_service(service_info)
    return zeroconf
```

Client

```
from zeroconf import Zeroconf, ServiceBrowser

class ChatServerListener:
    def add_service(self, zeroconf, type, name):
        info = zeroconf.get_service_info(type, name)
        if info:
            address = socket.inet_ntoa(info.addresses[0])
            port = info.port
            print(f"Server trovato: {address}:{port}")
            self.server_found(address, port)

def discover_mdns_server(self):
    zeroconf = Zeroconf()
    listener = ChatServerListener()
    browser = ServiceBrowser(zeroconf, "_chatserver._tcp.local.", listener)

    # Attende discovery...
```

Pro e Contro

□ Vantaggi

1. **Standard industriale**
    - Protocollo RFC ufficiale
    - Supporto nativo in molti OS
    - Ampiamente testato e affidabile
2. **Zero configuration**
    - Nessun setup richiesto
    - Plug and play reale
    - User-friendly
3. **Feature-rich**
    - Metadati servizio inclusi
    - Supporto multipli servizi
    - TXT records per info aggiuntive
    - Service browsing
4. **Efficienza**
    - Usa multicast intelligente
    - Cache distribuita

- Minimo traffico di rete

5. **Interoperabilità**

   - Funziona con servizi esistenti
   - Integrazione con Bonjour/Avahi
   - Standard cross-platform

□ Svantaggi

1. **Dipendenze esterne**

   - Richiede libreria `zeroconf` o `python-zeroconf`
   - Installazione dipendenze su ogni client
   - Potenziali problemi di versioning

2. **Complessità**

   - Curva di apprendimento ripida
   - Documentazione dispersa
   - Debugging complicato

3. **Limitazioni di rete**

   - Non attraversa router (come broadcast)
   - Problemi con alcune configurazioni firewall
   - Richiede multicast funzionante

4. **Overhead**

   - Libreria pesante per uso semplice
   - Consumo memoria maggiore
   - Startup time più lungo

5. **Non didattico**

   - Black box per studenti
   - Nasconde dettagli implementativi
   - Poco formativo su networking

Quando Usare mDNS

□ **Ideale per:**

- Applicazioni professionali
- Prodotti commerciali
- Integrazione con ecosistema Apple/Linux
- Necessità di standard riconosciuto
- Feature avanzate richieste

□ **Evitare quando:**

- Scopo didattico (troppo astratto)
- Dipendenze non accettabili
- Deployment semplificato necessario
- Controllo completo richiesto

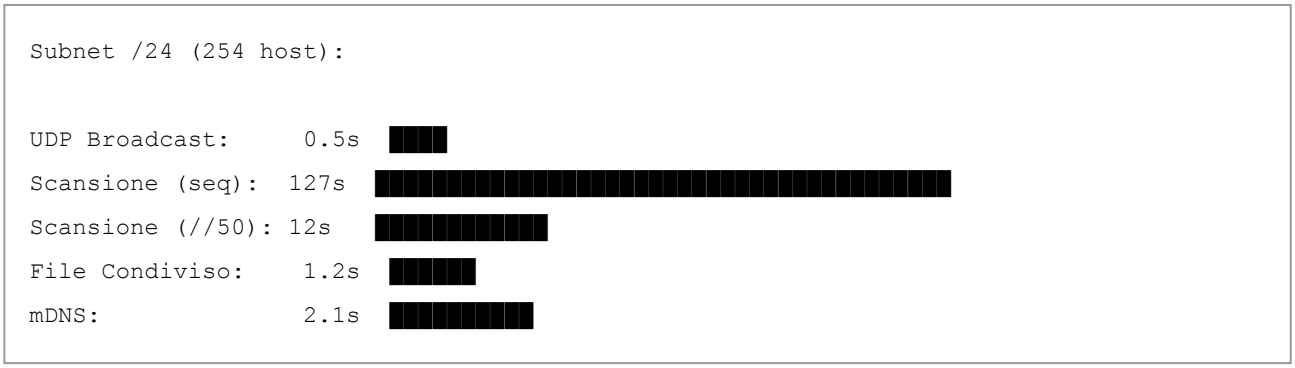
Confronto Comparativo

Tabella Comparativa Completa

Criterio	UDP Broadcast	Scansione Subnet	File Condiviso	mDNS/Zeroconf
Velocità	□□□□□ (<1s)	□□ (5-20s)	□□□□ (1-2s)	□□□□ (1-3s)
Affidabilità	□□□	□□□□□	□□□	□□□□
Semplicità Codice	□□□□	□□□	□□□□□	□□
Efficienza Rete	□□□□□	□□	□□□□□	□□□□
Compatibilità Firewall	□□	□□□□□	□□□□□	□□□
Scalabilità	□□□□□	□□	□□□	□□□□
Zero Config	□□□□□	□□□□□	□□	□□□□□
Valore Didattico	□□□□□	□□□□	□□	□□
Prod-Ready	□□□	□□□	□□	□□□□□
Setup Richiesto	Nessuno	Nessuno	Medio	Librerie
Dipendenze	Nessuna	Nessuna	Infrastruttura	python-zeroconf
Cross-Subnet	□ No	□ No	□ Sì	□ No
Sicurezza IDS	□ Ok	⚠ Port scan	□ Ok	□ Ok
Costo CPU	Basso	Alto	Basso	Medio
Costo Memoria	Basso	Medio	Basso	Alto

Metriche Prestazionali

Tempo di Discovery (Media)



Traffico di Rete Generato

```
UDP Broadcast:    ~500 bytes   ▌
Scansione /24:    ~15 KB       ████████████████████
File Condiviso:   0 bytes      (I/O file)
mDNS:             ~2 KB        ████
```

False Positive Rate

```
UDP Broadcast:    < 1%   (protocollo custom)
Scansione:        5-10%  (altri servizi su stessa porta)
File Condiviso:   < 1%   (lettura diretta)
mDNS:             < 1%   (service type specifico)
```

Matrice Decisionale

Caratteristiche Ambiente → Soluzione Raccomandata

Rete locale piccola Firewall permissivi Scopo didattico	UDP Broadcast □
Firewall restrittivi Wi-Fi guest Broadcast non funziona	Scansione Subnet (solo se < 50 host)
NAS/file server esiste Ambiente corporate Pochi client statici	File Condiviso (prototipi e test)
Prodotto commerciale Standard richiesto Integrazione Bonjour	mDNS/Zeroconf □

Raccomandazioni per Contesto Didattico

Per Corsi di Sistemi e Reti

Approccio Pedagogico Progressivo

Lezione 1: Concetti Base

- Problema: hardcoded IP addresses
- Introduzione al service discovery
- Soluzione manuale (file configurazione)

Lezione 2: UDP Broadcast □ RACCOMANDATO

- Differenza UDP vs TCP
- Concetto di broadcast
- Implementazione pratica
- Debug con Wireshark

Lezione 3: Scansione e Threading

- Port scanning etico
- Concurrency in Python
- ThreadPoolExecutor
- Ottimizzazioni performance

Lezione 4: Protocolli Standard

- Introduzione mDNS
- Analisi RFC
- Confronto soluzioni custom vs standard

Lab Pratico Suggerito

Esercitazione Completa (4 ore)

Parte 1: Implementazione Base (90 min)

1. Implementare chat server/client basico con IP hardcoded
2. Identificare limiti e problemi
3. Discussione: quali soluzioni esistono?

Parte 2: UDP Broadcast (90 min)

1. Aggiungere discovery service al server
2. Implementare discovery sul client
3. Testing in rete locale

4. Cattura traffico con Wireshark
5. Analisi pacchetti UDP

Parte 3: Problematiche Reali (45 min)

1. Simulare firewall: bloccare UDP broadcast
2. Tentare discovery → fallisce
3. Implementare fallback a scansione subnet
4. Confrontare prestazioni

Parte 4: Discussione e Ottimizzazioni (45 min)

1. Analisi pro/contro di ogni soluzione
2. Quando usare quale approccio
3. Sicurezza: autenticazione discovery
4. Progetti avanzati: load balancing, failover

Progetti Studente Avanzati

Livello Intermedio

- **Hybrid Discovery**: prova broadcast, poi scansione come fallback
- **Multi-Server Load Balancer**: client sceglie server meno carico
- **Server Heartbeat**: verifica periodica che server sia ancora attivo
- **Encrypted Discovery**: autenticazione tramite challenge-response

Livello Avanzato

- **Service Registry**: server centrale che traccia tutti i servizi
- **Geo-Discovery**: preferenza server geograficamente vicini
- **Fault Tolerance**: failover automatico se server primario cade
- **Cross-Subnet Discovery**: relay nodes per attraversare router

Valutazione e Criteri

Rubrica Valutazione Progetto

Criterio	Punti	Descrizione
Funzionalità	30	Discovery funziona in condizioni normali
Robustezza	20	Gestione errori e timeout
Performance	15	Velocità discovery accettabile
Codice	15	Leggibilità, commenti, struttura
Documentazione	10	README, diagrammi, esempi
Testing	10	Test cases e validazione

Domande di Verifica Comprensione

1. **Concettuali**
    - Perché UDP per discovery e TCP per chat?
    - Cosa succede se due server rispondono al broadcast?
    - Quali problemi causa il broadcast in reti grandi?
2. **Pratiche**
    - Come modificheresti il codice per supportare IPv6?
    - Come implementeresti retry con backoff esponenziale?
    - Come garantiresti che solo server autorizzati rispondano?
3. **Analisi**
    - Confronta overhead di broadcast vs scansione in subnet /16
    - Stima il tempo di scansione con 1000 host e timeout 0.5s
    - Quale soluzione useresti per IoT con 100+ dispositivi?

Risorse Didattiche Aggiuntive

Tools Consigliati

- **Wireshark**: analisi traffico di rete (broadcast, multicast)
- **nmap**: studio port scanning professionale
- **netcat**: testing manuale connessioni TCP/UDP
- **iperf**: misurazione performance di rete

Approfondimenti

- RFC 6762 (mDNS): lettura standard ufficiale
- Stevens "Unix Network Programming": capitoli su broadcast/multicast
- Beej's Guide to Network Programming: reference pratica
- Python `socket` documentation: API dettagliata

Progetti Open Source da Studiare

- **Avahi**: implementazione mDNS Linux
- **Bonjour**: implementazione Apple
- **Syncthing**: discovery P2P avanzato
- **BitTorrent DHT**: distributed hash table per discovery

Conclusioni e Best Practices

Decision Tree Finale

```
Devo implementare service discovery?
│
├─ Scopo didattico?
│  ├─ Sì → UDP Broadcast (insegna networking)
│  └─ No ↓
│
├─ Ambiente controllato (lab/ufficio)?
│  ├─ Sì → UDP Broadcast (veloce e semplice)
│  └─ No ↓
│
├─ Firewall bloccano broadcast?
│  ├─ Sì → Scansione Subnet (se < 100 host)
│  └─ No → UDP Broadcast
│
├─ Necessario attraversare router?
│  ├─ Sì → File Condiviso o Server Registry
│  └─ No → UDP Broadcast
│
├─ Prodotto commerciale?
│  └─ Sì → mDNS/Zeroconf (standard)
│
└─ Prototipo rapido interno?
   └─ Sì → File Condiviso (semplicissimo)
```

Checklist Implementazione

Prima di Iniziare

- [ ] Definire requisiti: velocità vs affidabilità
- [ ] Analizzare ambiente di rete target
- [ ] Verificare policy firewall
- [ ] Decidere se cross-subnet necessario
- [ ] Valutare competenze team

Durante Sviluppo

- [ ] Implementare timeout appropriati
- [ ] Gestire eccezioni di rete
- [ ] Aggiungere logging dettagliato
- [ ] Testare con Wireshark
- [ ] Validare su reti diverse

Prima del Deploy

- [ ] Test con firewall attivo
- [ ] Test con antivirus attivo
- [ ] Misurare performance reali
- [ ] Documentare requisiti di rete
- [ ] Preparare troubleshooting guide

Errori Comuni da Evitare

1. **Timeout troppo brevi**: miss dei server su reti lente
2. **Nessun fallback**: discovery fallisce, app inutilizzabile
3. **Mancanza logging**: debugging impossibile
4. **Ignorare IPv6**: sempre più comune
5. **Hardcoded subnet /24**: reti diverse usano /16, /22, ecc.
6. **Nessuna autenticazione**: rischi di sicurezza
7. **Broadcasting eccessivo**: congestione di rete
8. **Race conditions**: multipli server, nessuna sincronizzazione

Pattern Avanzati

Hybrid Approach (Raccomandato per Produzione)

```
def discover_server_robust(self):
    # 1. Prova cache locale
    if server := self.try_cache():
        return server

    # 2. Prova UDP broadcast (veloce)
    if server := self.udp_discovery(timeout=2):
        self.cache_server(server)
        return server

    # 3. Fallback a scansione (affidabile)
    if server := self.subnet_scan(max_workers=30):
        self.cache_server(server)
        return server

    # 4. Fallback manuale
    return self.manual_input()
```

Service Registry Pattern

```
# Server centrale mantiene lista servizi
class ServiceRegistry:
    def register(self, service_name, ip, port, ttl=300):
        self.services[service_name] = {
            'ip': ip,
            'port': port,
            'expires': time.time() + ttl
        }

    def discover(self, service_name):
        if service_name in self.services:
            if time.time() < self.services[service_name]['expires']:
                return self.services[service_name]
        return None
```

Sicurezza: Considerazioni Critiche

Autenticazione Discovery

```
import hmac
import hashlib

SECRET_KEY = b'shared-secret-key'

# Server
def authenticate_response(self, message):
    signature = hmac.new(SECRET_KEY, message.encode(), hashlib.sha256).hexdigest()
    return f"{message}|{signature}"

# Client
def verify_response(self, response):
    message, signature = response.rsplit('|', 1)
    expected = hmac.new(SECRET_KEY, message.encode(), hashlib.sha256).hexdigest()
    return hmac.compare_digest(signature, expected)
```

Rate Limiting

```
from collections import defaultdict
import time

class RateLimiter:
    def __init__(self, max_requests=5, window=60):
        self.requests = defaultdict(list)
        self.max_requests = max_requests
        self.window = window

    def allow_request(self, ip):
        now = time.time()
        # Rimuovi richieste vecchie
        self.requests[ip] = [t for t in self.requests[ip]
                             if now - t < self.window]

        if len(self.requests[ip]) < self.max_requests:
            self.requests[ip].append(now)
            return True
        return False
```

Riferimenti e Risorse

Standard e RFC

- **RFC 6762**: Multicast DNS
- **RFC 2782**: DNS SRV Records
- **RFC 919**: Broadcasting Internet Datagrams

Libri Consigliati

- "Unix Network Programming" - W. Richard Stevens
- "Computer Networks" - Andrew S. Tanenborough
- "Effective Python" - Brett Slatkin (per threading)

Documentazione Online

- Python `socket` module: https://docs.python.org/3/library/socket.html (https://docs.python.org/3/library/socket.html)
- Python `ipaddress` module: https://docs.python.org/3/library/ipaddress.html (https://docs.python.org/3/library/ipaddress.html)
- Zeroconf documentation: https://python-zeroconf.readthedocs.io/ (https://python-zeroconf.readthedocs.io/)

Tools e Software

- **Wireshark**: https://www.wireshark.org/ (https://www.wireshark.org/)
- **nmap**: https://nmap.org/ (https://nmap.org/)
- **Avahi**: https://www.avahi.org/ (https://www.avahi.org/)

Appendice: Codice Completo Implementazioni

A. UDP Broadcast Discovery (Versione Estesa)

```
# Vedi artifact chat_server_discovery
# Vedi artifact chat_client_discovery
```

B. Scansione Subnet (Versione Estesa)

```
# Vedi artifact chat_client_subnet_scan
```

C. Esempio File Condiviso

```
# server_config_writer.py
import json
import socket
from datetime import datetime

class ConfigFileServer:
    def __init__(self, config_path='\\\\shared\\server_config.json'):
        self.config_path = config_path
        self.port = 5000

    def register(self):
        config = {
            'ip': self.get_local_ip(),
            'port': self.port,
            'hostname': socket.gethostname(),
            'timestamp': datetime.now().isoformat(),
            'status': 'active'
        }

        try:
            with open(self.config_path, 'w') as f:
                json.dump(config, f, indent=2)
            print(f"Server registrato: {config}")
        except Exception as e:
            print(f"Errore registrazione: {e}")

    def get_local_ip(self):
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.connect(("8.8.8.8", 80))
        ip = s.getsockname()[0]
        s.close()
        return ip

# client_config_reader.py
import json
from datetime import datetime, timedelta

class ConfigFileClient:
    def __init__(self, config_path='\\\\shared\\server_config.json'):
        self.config_path = config_path

    def discover_server(self, max_age_minutes=5):
        try:
```

```
            with open(self.config_path, 'r') as f:
                config = json.load(f)

            # Verifica che il config non sia troppo vecchio
            timestamp = datetime.fromisoformat(config['timestamp'])
            age = datetime.now() - timestamp

            if age < timedelta(minutes=max_age_minutes):
                print(f"Server trovato: {config['ip']}:{config['port']}")
                return config['ip'], config['port']
            else:
                print(f"Config obsoleto (età: {age})")
                return None, None
        except FileNotFoundError:
            print("File di configurazione non trovato")
            return None, None
        except Exception as e:
            print(f"Errore lettura config: {e}")
            return None, None
```

D. Esempio mDNS/Zeroconf Base

```
# server_mdns.py
from zeroconf import Zeroconf, ServiceInfo
import socket
import time

class MDNSServer:
    def __init__(self, port=5000):
        self.port = port
        self.zeroconf = None
        self.service_info = None

    def register_service(self):
        self.zeroconf = Zeroconf()

        # Definizione servizio
        service_type = "_chatserver._tcp.local."
        service_name = f"MyChatServer.{service_type}"

        # Ottieni IP locale
        local_ip = self.get_local_ip()

        # Crea service info
        self.service_info = ServiceInfo(
            service_type,
            service_name,
            addresses=[socket.inet_aton(local_ip)],
            port=self.port,
            properties={
                b'version': b'1.0',
                b'description': b'Python Chat Server'
            },
            server=f"{socket.gethostname()}.local."
        )

        # Registra
        self.zeroconf.register_service(self.service_info)
        print(f"Servizio mDNS registrato: {service_name}")
        print(f"IP: {local_ip}, Porta: {self.port}")

    def unregister_service(self):
        if self.zeroconf and self.service_info:
            self.zeroconf.unregister_service(self.service_info)
            self.zeroconf.close()
```

```
            print("Servizio mDNS deregistrato")

    def get_local_ip(self):
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.connect(("8.8.8.8", 80))
        ip = s.getsockname()[0]
        s.close()
        return ip

# client_mdns.py
from zeroconf import Zeroconf, ServiceBrowser, ServiceStateChange
import socket
import time

class MDNSListener:
    def __init__(self):
        self.server_found = False
        self.server_ip = None
        self.server_port = None

    def on_service_state_change(self, zeroconf, service_type, name, state_change):
        if state_change is ServiceStateChange.Added:
            info = zeroconf.get_service_info(service_type, name)
            if info:
                self.server_ip = socket.inet_ntoa(info.addresses[0])
                self.server_port = info.port
                self.server_found = True
                print(f"Server trovato: {self.server_ip}:{self.server_port}")

class MDNSClient:
    def discover_server(self, timeout=5):
        zeroconf = Zeroconf()
        listener = MDNSListener()

        browser = ServiceBrowser(
            zeroconf,
            "_chatserver._tcp.local.",
            handlers=[listener.on_service_state_change]
        )

        # Attendi discovery
        start_time = time.time()
        while not listener.server_found and (time.time() - start_time) < timeout:
            time.sleep(0.1)
```

```
    zeroconf.close()

    if listener.server_found:
        return listener.server_ip, listener.server_port
    return None, None
```

Glossario Tecnico

- **Broadcast**: Invio di un messaggio a tutti i dispositivi in una subnet
- **Multicast**: Invio a un gruppo specifico di destinatari
- **Unicast**: Invio point-to-point a un singolo destinatario
- **Service Discovery**: Meccanismo per trovare servizi in rete automaticamente
- **mDNS**: Multicast DNS, protocollo per discovery locale
- **Subnet**: Sottorete, divisione logica di una rete IP
- **CIDR**: Classless Inter-Domain Routing, notazione per subnet (es. /24)
- **Port Scanning**: Tecnica per identificare porte aperte su un host
- **TTL**: Time To Live, durata di validità di un dato
- **Zeroconf**: Zero Configuration Networking, set di tecnologie per auto-config

---

Documento versione 1.0

Ultima modifica: Gennaio 2026

Autore: Sistema di documentazione tecnica