

NasoSanShare

Protocollo e architettura tecnica per condivisione file multi-dispositivo
(locale, remota, ibrida)

Documento	Specifica tecnica e logica di progetto
Versione documento	0.01
Piattaforme coinvolte	Windows (client), Android (client e server)
Componenti	App Client Windows, App Client Android, App Server Android
Analyst	NasoSan
Dev	Claude AI
Licenza	CC BY 4.0 - nasosan.it

Documento di specifica pre-sviluppo. Nessun codice e' stato ancora scritto. Le scelte indicate in questo documento (piattaforma, librerie, formati) sono proposte tecniche da approvare o modificare prima dell'avvio dello sviluppo, secondo il ciclo proposta-revisione-conferma.

Indice

1. Panoramica del progetto
2. Architettura generale
3. Componenti del sistema
4. Discovery locale (stessa rete)
5. Discovery remoto (reti diverse)
6. NAT Traversal e relay
7. Protocollo di trasferimento file
8. Formato dei messaggi (schema JSON)
9. Sicurezza
10. Struttura di progetto e convenzioni file
11. Stack tecnologico proposto
12. Fasi di sviluppo proposte
13. Decisioni aperte da confermare

1. Panoramica del progetto

NasoSanShare e' un sistema personale di condivisione file tra dispositivi (PC Windows, smartphone Android) che funziona in tre modalita', scelte automaticamente dall'app senza intervento manuale:

1. **Locale**: i due dispositivi sono sulla stessa rete WiFi. Si trovano da soli (discovery locale) e trasferiscono il file direttamente, senza passare da internet.
2. **Remota**: i due dispositivi sono su reti diverse (es. 4G e ADSL di casa). Si trovano tramite un server di discovery dedicato e tentano una connessione diretta punto-punto (P2P); se non riescono, i dati passano attraverso il server come relay.
3. **Mista**: l'app tenta sempre prima la via locale; se non trova nulla, passa automaticamente alla via remota. Nessun indirizzo va comunicato manualmente in nessun caso.

Il server di discovery/relay e' un'app dedicata destinata a un dispositivo Android fisso in casa (es. Xiaomi Mi 10T Pro sempre in carica su rete ADSL), separato dal PC Windows per non gravare sui suoi carichi di lavoro abituali.

2. Architettura generale

Schema logico dei tre scenari d'uso:

```
SCENARIO A - Stessa rete WiFi (locale)
[Client Windows] <---- discovery mDNS/UDP ----> [Client Android]
[Client Windows] <==== trasferimento diretto ==> [Client Android]
(nessun passaggio da internet, nessun server coinvolto)

SCENARIO B - Reti diverse, P2P riuscito (remoto diretto)
[Client A] ---- registrazione IP:porta ----> [Server Discovery Android]
[Client B] ---- registrazione IP:porta ----> [Server Discovery Android]
[Client A] <---- richiesta indirizzo B ----- [Server Discovery Android]
[Client A] <===== hole punching UDP =====> [Client B]
(il server serve solo a farli incontrare, poi si scollega dal flusso dati)

SCENARIO C - Reti diverse, P2P fallito (remoto via relay)
[Client A] =====> [Server Android: RELAY] =====> [Client B]
(i dati, cifrati end-to-end, rimbalzano sul server dedicato)
```

Regola generale: il server centrale (Server Discovery/Relay) non vede mai il contenuto dei file. Svolge due sole funzioni: 1) tenere un elenco dei dispositivi online con il loro indirizzo attuale, 2) fare da tramite cifrato quando il collegamento diretto non e' possibile.

3. Componenti del sistema

Componente	Piattaforma	Ruolo
App Client Windows	Windows 10/11	Invia/riceve file. Discovery locale + remoto. Nessun ruolo s
App Client Android	Android (telefono utente, es. Xiaomi Mi 14T)	Invia/riceve file. Discovery locale + remoto. Nessun ruolo s
App Server Android	Android dedicato (es. Mi 10T Pro, sempre acceso)	Discovery remoto (registro dispositivi online) + Relay (inolt

Nota: l'App Server Android e' concettualmente un servizio separato dall'App Client Android, anche se puo' condividere librerie comuni (crittografia, protocollo messaggi). Va eseguita come servizio Android in foreground persistente, distinta dall'app che l'utente usa normalmente per inviare/ricevere file.

4. Discovery locale (stessa rete)

Quando due dispositivi sono sulla stessa rete WiFi, si trovano senza alcun server, usando uno dei due meccanismi seguenti (da scegliere in fase di progetto, vedi tabella comparativa):

Meccanismo	Come funziona	Pro	Contro
mDNS / NSD (tipo Bonjour/Chromecast)	Ogni app si annuncia con un nome tipo nomeDispositivoLocalmente	Standard duplicato su tutti i dispositivi	Non Windows e Android (nessun pacchetto)
Broadcast/Multicast UDP custom	Ogni app invia periodicamente un pacchetto UDP con il proprio ID e tutte le informazioni pertinenti	Stesso codice su tutte le piattaforme	Non Windows e Android (nessun pacchetto)

Proposta: broadcast/multicast UDP custom, per avere lo stesso identico codice di discovery su Windows e Android (coerente con la regola di codice condiviso, nessuna logica duplicata tra piattaforme).

4.1 Formato pacchetto di annuncio (proposta)

Pacchetto UDP broadcast, inviato ogni 5 secondi sulla porta 47800 (porta dedicata, da confermare):

```
{
  "tipo": "annuncio",
  "idDispositivo": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
  "nomeDispositivo": "Xiaomi Mi 14T",
  "piattaforma": "android",
  "portaTrasferimento": 47810,
  "timestamp": "2026-08-02T10:15:00Z"
}
```

Ogni dispositivo mantiene una lista in memoria dei dispositivi visti negli ultimi 15 secondi (timeout); se un annuncio non arriva piu' entro quel tempo, il dispositivo viene rimosso dalla lista come non piu' raggiungibile in locale.

5. Discovery remoto (reti diverse)

Quando il discovery locale non trova l'altro dispositivo entro un tempo massimo (es. 3 secondi), l'app passa alla modalita' remota, contattando il Server Discovery Android.

5.1 Registrazione (client verso server)

Ogni client, all'avvio e periodicamente (keep-alive ogni 30-60 secondi), invia al server una registrazione con il proprio identificativo e l'indirizzo IP pubblico/porta rilevati automaticamente dal server stesso (il client non deve conoscere o comunicare il proprio IP: lo deduce il server dalla connessione in ingresso).

```
Richiesta client -> server (HTTPS o WebSocket persistente):
{
  "tipo": "registrazione",
  "idDispositivo": "xxxxxxxx-...",
  "nomeDispositivo": "PC Windows Studio",
  "piattaforma": "windows",
  "portaTrasferimentoLocale": 47810
}
```

Il server registra anche IP pubblico e porta sorgente osservati (non dichiarati dal client).

5.2 Richiesta indirizzo di un altro dispositivo

```
Richiesta client A -> server:
{ "tipo": "richiediIndirizzo", "idDispositivoTarget": "yyyyyyyy-..." }
```

```
Risposta server -> client A:
{
  "tipo": "indirizzoDispositivo",
  "idDispositivo": "yyyyyyyy-...",
  "online": true,
}
```

```
"ipPubblico": "203.0.113.45",  
"portaPubblica": 51122,  
"supportaP2P": true  
}
```

A questo punto A e B tentano l'hole punching (vedi capitolo 6). Il server resta in ascolto per coordinare lo scambio ma non partecipa al trasferimento, salvo passaggio in relay.

6. NAT Traversal e relay

6.1 Hole punching UDP (tentativo primario)

Il server comunica a entrambi i client l'indirizzo pubblico dell'altro. Nello stesso istante (coordinato dal server con un segnale di via), entrambi inviano pacchetti UDP l'uno verso l'altro. Sui NAT di tipo "cono" (la maggioranza dei router domestici e delle reti ADSL) questo apre un varco che permette la comunicazione diretta successiva.

6.2 Casi in cui l'hole punching fallisce

1. NAT simmetrico su uno o entrambi i lati (frequente su alcune reti dati mobili italiane).
2. Uno dei due dispositivi e' dietro una VPN (l'indirizzo visibile e' quello del server VPN, non quello reale, rendendo il punching inefficace nella pratica).
3. Firewall aziendali o restrittivi che bloccano traffico UDP non richiesto.

6.3 Fallback: relay tramite il Server Android

Se il P2P diretto non si stabilisce entro un timeout (es. 4-5 secondi), i client passano automaticamente in modalita' relay: aprono entrambi una connessione verso il Server Android (connessione in uscita, sempre permessa da qualunque rete) e il server inoltra i blocchi cifrati dall'uno all'altro in tempo reale, senza salvarli su disco.

Limite pratico del relay: la velocita' e' vincolata alla banda del Server Android e della sua rete ADSL di casa (upload in particolare, spesso il collo di bottiglia sulle linee ADSL italiane). Il P2P diretto resta quindi sempre la via preferita quando disponibile.

6.4 Raggiungibilita' del Server Android

Il Server Android deve avere una porta raggiungibile dall'esterno per il ruolo di relay/discovery. Serve una regola di port forwarding sul router di casa (una tantum, in fase di installazione), che inoltri la porta scelta (es. 47900 TCP) verso l'IP locale del telefono. In alternativa, se il router lo supporta, si puo' usare UPnP per automatizzare l'apertura della porta senza intervento manuale.

7. Protocollo di trasferimento file

Il protocollo di trasferimento vero e proprio e' identico indipendentemente dal fatto che la connessione sia locale diretta, remota diretta (P2P) o remota via relay: cambia solo il tramite, non la logica.

7.1 Fasi del trasferimento

1. **Handshake**: il mittente invia un pacchetto di controllo con nome file, dimensione totale, hash SHA-256 dell'intero file, numero di blocchi previsti. Il destinatario risponde con accettazione o rifiuto (es. spazio insufficiente).

2. **Scambio chiave:** se non già stabilito da una sessione precedente, i due dispositivi negoziano una chiave di sessione tramite Diffie-Hellman (o si usa TLS se il trasporto lo supporta nativamente).
3. **Invio a blocchi (chunk):** il file viene suddiviso in blocchi di dimensione configurabile (proposta: 256 KB), ciascuno cifrato singolarmente e accompagnato dal proprio hash e numero progressivo.
4. **Conferma per blocco:** il destinatario conferma la ricezione corretta di ogni blocco (o richiede il reinvio se l'hash non corrisponde).
5. **Verifica finale:** a trasferimento completo, il destinatario ricalcola l'hash dell'intero file ricostruito e lo confronta con quello dichiarato nell'handshake.
6. **Ripresa dopo interruzione:** se la connessione cade, alla riconnessione il destinatario comunica l'elenco dei blocchi già ricevuti; il mittente invia solo quelli mancanti.

7.2 Cifratura

Cifratura end-to-end obbligatoria su ogni percorso, incluso il relay: il Server Android inoltra sempre e solo byte già cifrati dal mittente, senza possibilità di leggerne il contenuto. Algoritmo proposto: AES-256-GCM per i blocchi dati, con chiave di sessione derivata da scambio Diffie-Hellman (curva ellittica, es. X25519) eseguito una sola volta all'inizio della sessione tra i due dispositivi.

8. Formato dei messaggi (schema JSON)

Tutti i messaggi di controllo (discovery, handshake, conferme) usano JSON su UTF-8 senza BOM, coerente con la convenzione di codifica file JSON già in uso. I blocchi dati binari (contenuto file cifrato) viaggiano separatamente dal JSON, con un'intestazione binaria minima (lunghezza + numero blocco + hash).

Campo	Tipo	Descrizione
tipo	stringa	Tipo messaggio: annuncio, registrazione, richiestaIndirizzo, indirizzoDispositivo, handshake
idDispositivo	stringa (UUID)	Identificativo univoco del dispositivo, generato alla prima installazione e conservato in c
idSessione	stringa (UUID)	Identifica un trasferimento in corso, per gestire ripresa e blocchi multipli in parallelo
timestamp	stringa (ISO 8601)	Data/ora del messaggio, per gestione timeout e log

9. Sicurezza

1. Nessun dato sensibile (chiavi, ID dispositivo) memorizzato in chiaro: chiavi di sessione tenute solo in memoria, mai scritte su disco.
2. Il Server Android non ha accesso al contenuto dei file in nessuna modalità (locale, P2P, relay): vede solo blocchi cifrati e metadati minimi (dimensione, numero blocchi).
3. Autenticazione tra dispositivi: al primo abbinamento (pairing) tra due dispositivi, scambio di una chiave pubblica fissa (associata all'idDispositivo), così i trasferimenti successivi verificano che l'interlocutore sia davvero il dispositivo atteso e non un impostore in ascolto sulla stessa rete o sul server.
4. Elenco dispositivi autorizzati: solo dispositivi esplicitamente abbinati dall'utente (whitelist locale su ogni client) possono inviare/ricevere file; il server di discovery si limita a farli incontrare ma non autorizza nulla.
5. Log del Server Android: registra solo eventi di connessione (chi si è connesso, quando, esito), mai contenuti o nomi file, coerente con la regola di non registrare dati sensibili.

10. Struttura di progetto e convenzioni file

Struttura cartelle secondo lo standard in uso (script\progetto\...), applicata ai tre sotto-progetti:

```
script\  
  indice_logiche.json  
  NasoSanShareClientWindows\  
    log\  cfg\  media\  
    app.json (contiene il campo "versione")  
    indice_file.md  
    logica_NasoSanShareClientWindows.md  
    NasoSanShareClientWindows.py (base)  
    NasoSanShareClientWindows1.py ... N.py (sezioni)  
  NasoSanShareClientAndroid\  
    log\  cfg\  media\  
    app.json  
    indice_file.md  
    logica_NasoSanShareClientAndroid.md  
    [struttura Kotlin standard: package com.nasosan.share.client]  
  NasoSanShareServerAndroid\  
    log\  cfg\  media\  
    app.json  
    indice_file.md  
    logica_NasoSanShareServerAndroid.md  
    [struttura Kotlin standard: package com.nasosan.share.server]
```

Ogni progetto mantiene versione indipendente in proprio app.json (es. Client Windows 0.010, Client Android 0.005, Server Android 0.015): non e' un'unica versione condivisa, perche' sono tre applicazioni distinte con cicli di rilascio propri. Il protocollo di comunicazione tra loro (formato messaggi, capitoli 7-8) ha una propria versione di compatibilita' dichiarata in ogni handshake, per evitare che un client aggiornato non riesca piu' a parlare con un server non ancora aggiornato.

11. Stack tecnologico proposto

Componente	Linguaggio/Framework proposto	Motivazione
Client Android	Kotlin (coerente con NasoSanPlayer)	Stesso linguaggio gia' in uso, NsdManager nativo per mDNS, accesso diretto a file
Server Android	Kotlin, servizio Foreground Service persistente	Stessa base di codice del client per le parti comuni (protocollo, cifratura); coerenza
Client Windows	Python (asyncio) con interfaccia grafica Tkinter, oppure C#/.NET stack WinForms (PowerShell/Python) e con le librerie note (asyncio, cryptography)	Python, oppure C#/.NET stack WinForms

Da decidere: Python vs C#/.NET per il client Windows. Il Python permette di riusare esperienza e librerie gia' note (asyncio, cryptography); il C#/.NET produce un eseguibile piu' leggero e senza dipendenza da un interprete installato, ma richiede di lavorare in un linguaggio non ancora usato nei progetti precedenti.

11.1 Librerie chiave per funzione

Funzione	Kotlin (Android)	Python (Windows)
Discovery locale UDP	DatagramSocket / NsdManager	modulo socket (libreria standard)
Cifratura	javax.crypto (AES-GCM), libreria per X.509	cryptography
Comunicazione server (HTTPS/WebSocket)	OkHttp + WebSocket	libreria websockets o aiohttp
Interfaccia grafica	Jetpack Compose o layout XML standard	Inter (o WinForms se si sceglie C#/.NET)

12. Fasi di sviluppo proposte

1. **Fase 1 - Protocollo base:** definizione definitiva del formato messaggi e cifratura, implementata prima come libreria condivisa testabile in isolamento (senza GUI).
2. **Fase 2 - Discovery locale:** client Windows e Android capaci di trovarsi sulla stessa rete e trasferire un file di test, senza server.
3. **Fase 3 - Server Android:** app server con registrazione dispositivi e richiesta indirizzo, senza ancora relay (solo scambio di indirizzi per hole punching).
4. **Fase 4 - Hole punching remoto:** trasferimento diretto tra reti diverse quando il NAT lo permette.
5. **Fase 5 - Relay:** fallback funzionante quando il P2P fallisce, con misura reale delle prestazioni sulla rete ADSL di casa.
6. **Fase 6 - Interfacce definitive:** rifinitura GUI di entrambi i client (selezione file, progress bar, cronologia trasferimenti, gestione dispositivi abbinati).
7. **Fase 7 - Gestione errori e robustezza:** riconnessione automatica, ripresa trasferimenti interrotti, gestione riavvio del Server Android (autostart del servizio dopo reboot del telefono).

13. Decisioni aperte da confermare

1. Meccanismo di discovery locale: mDNS/NSD nativo oppure broadcast UDP custom (proposta: custom, per codice condiviso tra piattaforme).
2. Linguaggio del Client Windows: Python o C#/.NET.
3. Porta/e da riservare per discovery locale, discovery remoto e relay (proposta iniziale: 47800 locale, 47900 remoto/relay, da verificare contro conflitti con altri servizi già in uso, es. Agent DVR, Tailscale).
4. Dimensione del blocco (chunk) per il trasferimento file (proposta iniziale: 256 KB, regolabile).
5. Se implementare anche il fallback UPnP per l'apertura automatica della porta sul router, oltre al port forwarding manuale.
6. Se prevedere sincronizzazione automatica di una cartella (tipo Syncthing) oppure solo invio manuale di file su richiesta (più semplice da realizzare in una prima versione).