Building a Container in C from Scratch
for FUN and PROFIT

Roberto Marino, PhD

Università di Messina
CdL in Informatica — Reti e Sistemi Distribuiti Mod.B

April 15, 2026



UniMe
1548

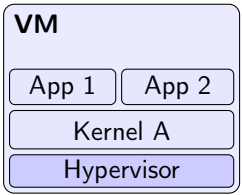
Il container non è una VM

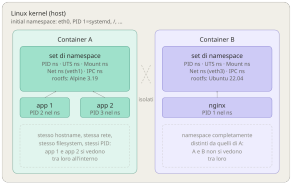
Virtual Machine

- Kernel **separato** per ogni VM
- Hypervisor emula hardware
- Isolamento forte ma pesante
- Boot: secondi/minuti
- Overhead: ∼100–500 MB RAM

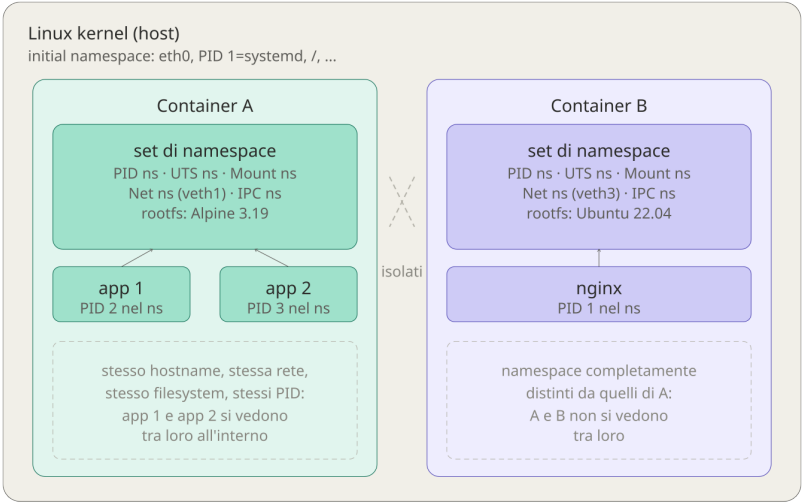
Container

- **Stesso kernel** dell'host
- Solo *isolamento* a livello kernel
- Leggero, avvio in millisecondi
- Overhead: ∼1–5 MB RAM
- Implementato con **namespaces** + **cgroups**





Container: un astrazione che vive all'interno del SO



Cosa sono i namespace?

Definizione

Un **namespace** è un meccanismo del kernel Linux che *isola* una risorsa di sistema: ogni processo vede la *propria* vista di quella risorsa, indipendentemente dagli altri.

Oggi esistono 8 tipi di namespace:

Flag	Namespace	Cosa isola
CLONE_NEWPID	PID	Albero dei processi
CLONE_NEWUTS	UTS	Hostname, domainname
CLONE_NEWNS	Mount	Filesystem e mount point
CLONE_NEWNET	Net	Stack di rete, interfacce, porte
CLONE_NEWIPC	IPC	Code messaggi, semafori, shared mem
CLONE_NEWUSER	User	UID/GID mapping
CLONE_NEWCGROUP	Cgroup	Vista del cgroup filesystem
CLONE_NEWTIME	Time	Clock MONOTONIC e BOOTTIME

clone() — definizione e parametri

definizione

```
#include <sched.h>

pid_t clone(
    int   (*fn)(void *), /* funzione figlio  */
    void   *stack,       /* top dello stack  */
    int     flags,       /* cosa isolare     */
    void   *arg,         /* argomento a fn() */
    /* pid_t *ptid, void *tls, pid_t *ctid */
);
/* ritorna: PID del figlio nel parent,
            fn() parte direttamente */
```

Param.	Significato
`fn`	Prima funzione eseguita dal figlio
`stack`	Cima dello stack (x86-64: cresce verso il basso, passare `buf+size`)
`flags`	OR di `CLONE_*`: namespace da creare e risorse da condividere
`arg`	Puntatore opaco passato come unico argomento a `fn()`

Flag — namespace (isolano)

@IX@

Flag	Crea nuovo namespace per...
`CLONE_NEWPID`	albero dei processi
`CLONE_NEWUTS`	hostname e domainname
`CLONE_NEWNS`	mount point e filesystem
`CLONE_NEWNET`	interfacce, porte, routing
`CLONE_NEWIPC`	code msg, semafori, shm
`CLONE_NEWUSER`	UID/GID (rootless container)
`CLONE_NEWCGROUP`	vista del cgroup fs

Flag — condivisione (per thread)

@IX@

Flag	Condivide con il parent...
`CLONE_VM`	spazio di memoria (heap, stack)
`CLONE_FILES`	tabella file descriptor
`CLONE_FS`	cwd, root, umask
`CLONE_SIGHAND`	signal handler
`CLONE_THREAD`	thread group (stesso PID esterno)

Stack manuale

Come si crea un namespace con `clone()`

Codice del container

```
int flags = CLONE_NEWPID  /* nuovo albero PID    */
          | CLONE_NEWUTS  /* hostname proprio    */
          | CLONE_NEWNS   /* mount namespace     */
          | CLONE_NEWNET  /* stack rete vuoto    */
          | CLONE_NEWIPC  /* IPC/shm separati    */
          | SIGCHLD;      /* notifica parent     */

char *stack     = malloc(1024 * 1024);
char *stack_top = stack + 1024 * 1024;

pid_t child = clone(child_fn,
                    stack_top,
                    flags,
                    &args);
/* child_fn() parte gia' nei nuovi ns */
```

Atomicita'

Il kernel crea tutti i namespace elencati nei flag in un'unica operazione, prima che `child_fn()` esegua la sua prima istruzione.

Non esiste una finestra in cui il figlio e' a meta' isolato.

Cosa vede child_fn()

```
static int child_fn(void *arg) {
  /* siamo gia' in tutti i ns nuovi */

  /* UTS ns: ok cambiare hostname */
  sethostname("container", 9);

  /* PID ns: questo processo e' PID 1 */
  printf("%d\n", getpid());  /* -> 1 */

  /* Net ns: solo lo, nessuna eth0   */

  /* Mount ns: stessa vista dell'host
     finche' non eseguiamo pivot_root */
  execvp("/bin/sh", ...);
  return 1;
}
```

Verifica dall'host

```
# i namespace del figlio sono diversi:
ls -la /proc/<child_pid>/ns/
# pid -> pid:[4026532193]  # diverso!
# uts -> uts:[4026532194]
```

clone() vs fork()

fork() — il classico

```
pid_t pid = fork();
if (pid == 0) {
    /* figlio:
       - stessa riga di codice del parent
       - stesso PID namespace
       - stessa rete, stesso filesystem
       - memoria separata (copy-on-write)
       - file descriptor copiati */
    printf("sono figlio, pid=%d\n",
           getpid());
    exit(0);
} else {
    /* parent riceve PID del figlio */
    waitpid(pid, NULL, 0);
}
```

`fork()` non puo' creare nuovi namespace. Il figlio vede lo stesso mondo del parent. Internamente: `clone(SIGCHLD)` senza altri flag.

clone() — controllo totale

```
/* con namespace: container isolato */
pid_t pid = clone(child_fn,
    stack_top,
    CLONE_NEWPID | CLONE_NEWNS |
    CLONE_NEWNET | SIGCHLD,
    &args);

/* senza namespace: come fork()
   ma con funzione dedicata */
pid_t pid = clone(child_fn,
    stack_top,
    SIGCHLD,
    &args);
```

@IXX@

fork()	clone()
Namespace	stessi del parent	nuovi con `CLONE_NEW*`
Stack	copiato (CoW)	allocato manualmente
Ingresso	stessa riga	funzione dedicata
Ritorno	0 / PID figlio	PID figlio (sempre)
Memoria	separata (CoW)	separata (o `CLONE_VM`)
Uso tipico	processi UNIX	container, thread

pthread_create() vs fork()

fork() — processi separati

```
int x = 42;

pid_t pid = fork();
if (pid == 0) {
    x = 100;          /* copia del parent */
    exit(0);
}
waitpid(pid, NULL, 0);
printf("%d\n", x);    /* stampa 42 */
/* il figlio ha modificato la SUA copia
   della variabile, non quella del parent */

/* comunicazione: pipe, socket, shm */
int pipefd[2];
pipe(pipefd);
write(pipefd[1], &x, sizeof(x));
```

pthread_create() — thread condivisi

```
/* pthread_create chiama clone() con:    */
/* CLONE_VM|CLONE_FS|CLONE_FILES|        */
/* CLONE_SIGHAND|CLONE_THREAD|...        */

int x = 42;
pthread_t t;
pthread_create(&t, NULL, fn, NULL);

void *fn(void *arg) {
    x = 100;    /* modifica DIRETTA */
    /* stessa memoria, stessi fd,
       stesso PID esterno (thread group) */
    return NULL;
}
pthread_join(t, NULL);
printf("%d\n", x);  /* stampa 100 */
```

Isolamento

Crash del figlio ⇒ non tocca il parent.
Il kernel separa completamente la memoria.

PID diverso, file descriptor copiati.

@IXX@

fork()	pthread_create()
Memoria	separata (CoW) condivisa
File desc.	copia condivisi
PID esterno	nuovo stesso del parent
Crash	isolato abbatte tutto
Comunic.	IPC/pipe/socket variabili, mutex

unshare() e setns() — gli altri due modi

unshare() — nuovi namespace, stesso processo

```c
#include <sched.h>

/* Crea nuovi namespace per il processo
   CORRENTE senza creare un figlio.
   Il processo si "stacca" dalle risorse
   condivise col parent. */

unshare(CLONE_NEWNS);    /* mount ns  */
unshare(CLONE_NEWNET);   /* net ns    */
unshare(CLONE_NEWPID);   /* pid ns    */

/* combinabili in una sola chiamata: */
unshare(CLONE_NEWNS | CLONE_NEWNET);

/* comando shell equivalente: */
/* unshare --pid --mount /bin/sh */
```

clone() vs unshare()

`clone()` **crea un figlio** nei nuovi namespace.
`unshare()` **sposta il processo corrente** in nuovi namespace.

setns() — entra in un namespace esistente

```c
#include <sched.h>
#include <fcntl.h>

/* Entra nel namespace di un processo
   gia' in esecuzione.
   Usato da: docker exec, nsenter */

/* 1. apri il file descrittore del ns */
int fd = open(
    "/proc/<container_pid>/ns/net",
    O_RDONLY);

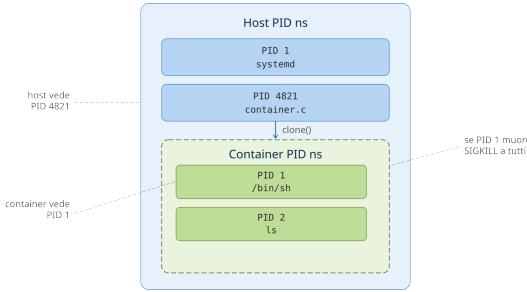
/* 2. entra nel namespace */
setns(fd, CLONE_NEWNET);
close(fd);

/* ora siamo nella rete del container */
execvp("/bin/sh", ...);
```

nsenter dalla shell

`PID namespace` — Isolamento dei processi

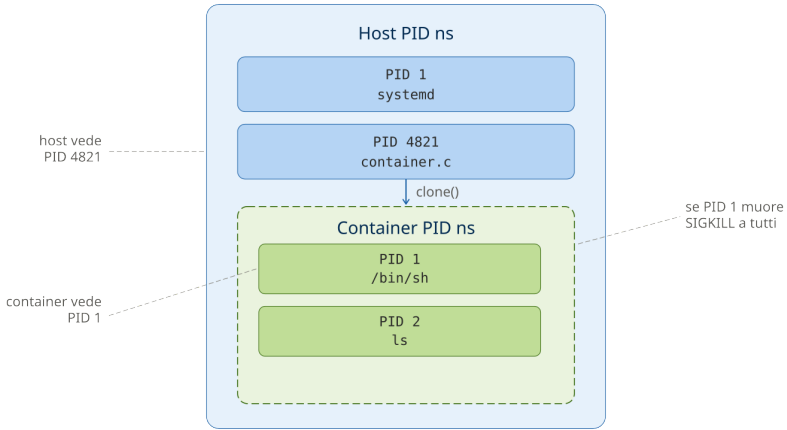
- Ogni container ha il suo **albero PID** indipendente
- Il primo processo del container è **PID 1** nel suo namespace
- L'host vede il vero PID (es. 4821), il container vede PID 1
- Un processo non può "vedere" i processi fuori dal suo namespace
- **Init semantics**: se PID 1 muore, il kernel manda SIGKILL a tutti i processi del namespace



Verifica

```
# dentro il container:
ps aux
# mostra solo i tuoi processi
# PID 1 = /bin/sh
```

Pid namespace



UTS namespace — Hostname e domainname

UTS = UNIX Time-sharing System (nome storico della struttura kernel che contiene hostname e domainname)

- Permette ad ogni container di avere un **hostname distinto**
- Non impatta le altre risorse, ma è essenziale per l'identità del container
- Usato da Docker per assegnare nomi come `a3f9d1bc...`
- La syscall è `sethostname()`

Dal nostro codice:

```
/* child_fn(), dopo clone() con CLONE_NEWUTS */
if (sethostname(HOSTNAME, strlen(HOSTNAME)) < 0) {
    perror("sethostname");
    return 1;
}
/* Ora il container ha hostname "container"
   senza modificare l'hostname dell'host */
```

Demo

```
# sull'host:
$ hostname
my-laptop

# dentro il container:
$ hostname
container

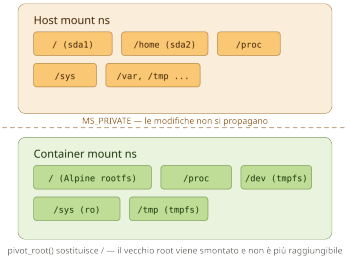
# sull'host (non cambia):
$ hostname
my-laptop
```

`Mount namespace` — Filesystem isolato

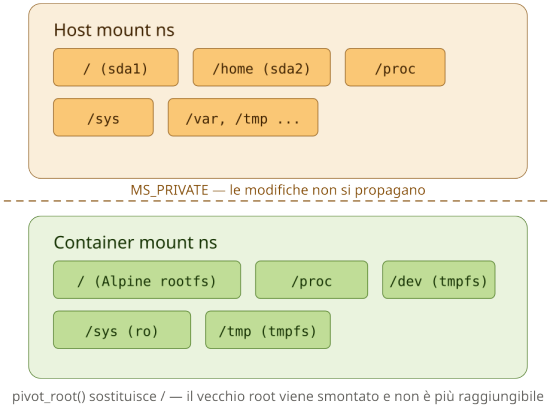
- Ogni container ha la sua **vista del filesystem**
- I mount fatti dentro *non si propagano* all'host (e viceversa)
- Prerequisito per `pivot_root()`: rende il rootfs del container visibile solo dentro
- Prima operazione: rendere `/` **`MS_PRIVATE`**

```
/* Rende il mount namespace privato:
   nessun evento di mount si propaga */
mount(NULL, "/", NULL,
      MS_REC | MS_PRIVATE, NULL);

/* Ora monta proc, sysfs, devtmpfs
   solo nel container */
mount("proc", "/proc", "proc",
      MS_NOSUID|MS_NOEXEC|MS_NODEV,
      NULL);
```



Mount namespace



pivot_root() sostituisce / — il vecchio root viene smontato e non è più raggiungibile

pivot_root() — La syscall che cambia il root

Differenza tra chroot() e pivot_root()

chroot() cambia solo / per il processo corrente, ma il vecchio root è ancora accessibile da un processo privilegiato (jail escape classico). pivot_root() è un'operazione atomica a livello kernel: sostituisce il root e *smonta* il vecchio.

```
static int pivot_root_to(const char *new_root) {
    char old_root[256];
    snprintf(old_root, sizeof(old_root), "%s/.pivot_old", new_root);

    /* Il new_root DEVE essere un mount point (bind mount) */
    mount(new_root, new_root, NULL, MS_BIND | MS_REC, NULL);
    mkdir(old_root, 0700);

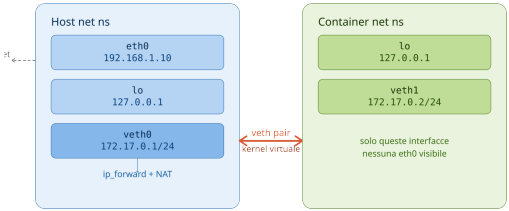
    /* Non c'e' wrapper glibc: usiamo syscall() direttamente */
    syscall(SYS_pivot_root, new_root, old_root);

    chdir("/");                          /* ora siamo nel nuovo root */
    umount2("/.pivot_old", MNT_DETACH);  /* smonta il vecchio root  */
    rmdir("/.pivot_old");
    return 0;
}
```

Nota: pivot_root() non ha wrapper in glibc — va chiamata con syscall()(SYS_pivot_root, ...).

`Net namespace` — Stack di rete isolato

- Ogni container ha le sue **interfacce di rete**, tabelle di routing, regole iptables, porte
- Un nuovo net namespace ha solo `lo` (loopback) e nient'altro
- Per connettere container ↔ host si usano **veth pair** (virtual ethernet)
- Docker crea un bridge `docker0` e una `veth` per container

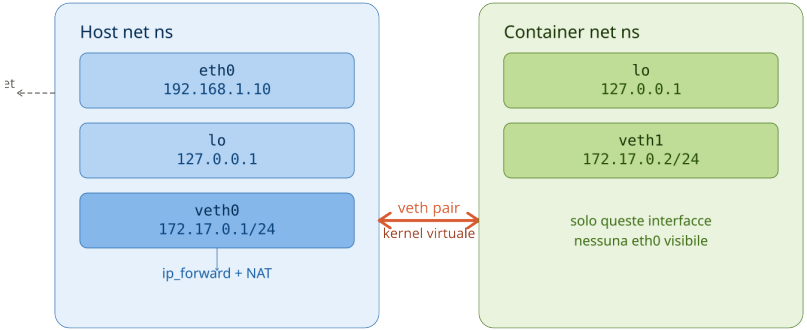


Verifica

```
# dentro il container (solo CLONE_NEWNET):
ip link show
# 1: lo: <LOOPBACK> ...
# Nessuna eth0, nessuna connessione!

# Sull'host rimane eth0 intatta
```

Net namespace



`IPC namespace` — Comunicazione inter-processo

- Isola le risorse **IPC System V**: code messaggi (`msgget`), semafori (`semget`), segmenti di shared memory (`shmget`)
- Isola anche le **POSIX message queues** (`mq_open`)
- Senza questo namespace, un processo del container potrebbe leggere/scrivere su una shm dell'host con lo stesso key
- Spesso trascurato ma importante per la sicurezza

Esempio concreto

```
# Sull'host:
$ ipcmk -M 4096
Shared memory id: 327684

# Nel container (CLONE_NEWIPC):
$ ipcs -m
# Nessuna shared memory!
# Namespace separato

# Senza CLONE_NEWIPC:
$ ipcs -m
# Vede la shm 327684!  <- PERICOLOSO
```

Nel nostro `container.c`: il flag `CLONE_NEWIPC` è incluso nella chiamata a `clone()`.

Il namespace più potente (e delicato)

Permette a un processo di essere `root` (UID 0) *dentro* il namespace, mappato a un UID non privilegiato fuori. Questo è ciò che rende possibili i **container rootless**.

```
/* Dopo clone() con CLONE_NEWUSER, il parent scrive le mappe: */

/* /proc/<child_pid>/uid_map : uid_inside uid_outside count */
/* Esempio: il container vede UID 0, che e' UID 1000 sull'host */
write(uid_map_fd, "0 1000 1", 8);

/* /proc/<child_pid>/gid_map : idem per GID */
write(gid_map_fd, "0 1000 1", 8);
```

Vantaggi:

- Nessun sudo per avviare container
- Root nel container $\neq$ root sull'host
- Usato da Podman (rootless by default)

Nel nostro codice:

Non implementato per semplicità, ma è il passo successivo. Il flag è `CLONE_NEWUSER` e richiede di scrivere `/proc/<pid>/uid_map` e `gid_map` dal parent.

Cgroup namespace e Time namespace

Cgroup namespace (Linux 4.6)

- Un container vede / come la root del suo cgroup
- Senza questo ns, il container vede il path completo sull'host: `/sys/fs/cgroup/container_demo/...`
- Con questo ns vede solo /
- Usato insieme a `CLONE_NEWNS` per nascondere la struttura cgroup dell'host

Time namespace (Linux 5.6)

- Il più recente (2020, Linux 5.6)
- Isola `CLOCK_MONOTONIC` e `CLOCK_BOOTTIME`
- Permette a container migrati di mantenere un orologio coerente
- Utile per live migration e checkpoint/restore (**CRIU**)
- `CLOCK_REALTIME` rimane globale

cgroups v2 — Control Groups

Cosa sono

I **cgroups** (control groups) limitano e misurano le risorse usate da un gruppo di processi. Sono il complemento dei namespace: i namespace nascondono, i cgroups *limitano*.

Cosa si può limitare:

- `memory.max` — RAM massima
- `memory.swap.max` — Swap massima
- `cpu.max` — Quota CPU
- `pids.max` — N. di processi
- `io.max` — I/O su disco
- `cpuset.cpus` — Core specifici

Come funziona (cgroups v2):

1. Crea directory in `/sys/fs/cgroup/`
2. Scrivi i limiti nei file di controllo
3. Scrivi il PID del processo in `cgroup.procs`
4. Il kernel applica i limiti automaticamente

cgroups v2 — Il codice

Listing 1: setup_cgroup() da container.c

```
static int setup_cgroup(pid_t child_pid) {
    char path[256], pid_str[32];

    /* 1. Crea la directory del cgroup */
    mkdir(CGROUP_BASE, 0755);   /* /sys/fs/cgroup/container_demo */

    /* 2. Limita la memoria a 64 MB */
    snprintf(path, sizeof(path), CGROUP_BASE "/memory.max");
    write_file(path, "67108864");

    /* 3. Blocca lo swap */
    snprintf(path, sizeof(path), CGROUP_BASE "/memory.swap.max");
    write_file(path, "0");

    /* 4. Massimo 32 processi */
    snprintf(path, sizeof(path), CGROUP_BASE "/pids.max");
    write_file(path, "32");

    /* 5. Assegna il container al cgroup */
    snprintf(path, sizeof(path), CGROUP_BASE "/cgroup.procs");
    snprintf(pid_str, sizeof(pid_str), "%d", child_pid);
    write_file(path, pid_str);   /* ora i limiti sono attivi */
    return 0;
}
```

Nota: La configurazione avviene nel *parent*, dopo `clone()` ma prima che il figlio chiami execvp().

clone() — Il cuore del container

Listing 2: La chiamata clone() da container.c

```
int flags = CLONE_NEWPID   /* nuovo albero PID              */
          | CLONE_NEWUTS   /* hostname indipendente         */
          | CLONE_NEWNS    /* mount namespace isolato       */
          | CLONE_NEWNET   /* stack di rete vuoto           */
          | CLONE_NEWIPC   /* IPC/shm separati              */
          | SIGCHLD;       /* segnala al parent alla morte  */

/* clone() = fork() + creazione namespace + stack custom  */
pid_t child_pid = clone(
    child_fn,    /* funzione eseguita dal figlio          */
    stack_top,   /* top dello stack allocato (1 MB)       */
    flags,       /* namespace da creare                   */
    &args        /* argomenti passati a child_fn          */
);
```

Differenza con `fork()`:

- `fork()`: copia il processo, stesso namespace
- `clone()`: crea nuovi namespace atomicamente
- Lo stack viene allocato manualmente

Perché `stack_top`?

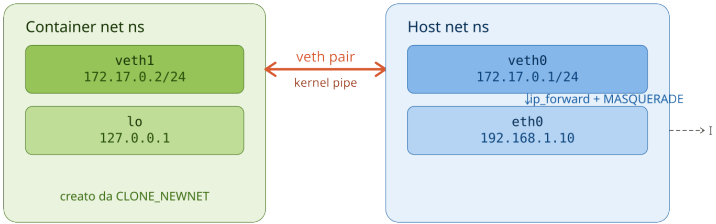
Gli stack crescono verso indirizzi *bassi* su x86-64. Quindi si passa il *puntatore alla cima* dell'area allocata:

```
char *stack = malloc(1024*1024);
char *stack_top = stack + 1024*1024;
/* stack_top e' l'indirizzo piu' alto */
```

Perché il container non ha rete (di default)

Con `CLONE_NEWNET` il container ottiene un net namespace **completamente vuoto**: solo l'interfaccia `lo` (loopback), nessuna route, nessun accesso all'esterno.

Per connettere container e host usiamo una **veth pair**: due interfacce virtuali collegate come un tubo — tutto ciò che entra da un lato esce dall'altro.



I passi necessari — chi fa cosa

La configurazione è **divisa** tra parent (host) e child (container):

Parent (host) — `setup_veth()`

1. **Crea** la coppia `veth0` ↔ `veth1` nel proprio net ns (`RTM_NEWLINK`, kind=`"veth"`)
2. **Sposta** `veth1` nel net ns del container (`IFLA_NET_NS_PID`)
3. Porta `veth0` in stato **UP** (`ioctl SIOCSIFFLAGS`)
4. Assegna `172.17.0.1/24` a `veth0` (`RTM_NEWADDR`)

Child (container) — `setup_container_net()`

1. Porta `lo` in stato **UP**
2. Porta `veth1` in stato **UP**
3. Assegna `172.17.0.2/24` a `veth1`

Il child vede già `veth1` nel suo net ns perché il parent l'ha spostata prima che `execvp()` fosse chiamata.

Per abilitare Internet (opzionale, post-avvio)

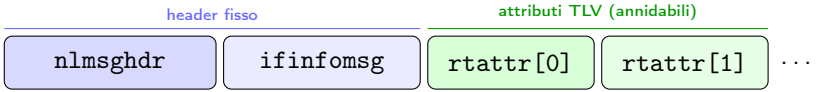
```
echo 1 > /proc/sys/net/ipv4/ip_forward
iptables -t nat -A POSTROUTING -s 172.17.0.0/24 -o eth0 -j MASQUERADE
```

Netlink — come si parla al kernel in C

Cos'è Netlink

Netlink è il canale ufficiale kernel → userspace per configurare la rete. Invece di eseguire `ip link` in una shell, apriamo un socket `AF_NETLINK` e inviamo messaggi strutturati direttamente al kernel.

Ogni messaggio è composto da:



Listing 3: Apertura socket e invio con ACK

```
int nl_fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
bind(nl_fd, (struct sockaddr *)&sa, sizeof(sa));

/* ... costruisci nlh con nlmsg_type = RTM_NEWLINK ... */

send(nl_fd, nlh, nlh->nlmsg_len, 0);

/* Il kernel risponde con NLMSG_ERROR: error==0 = successo */
recv(nl_fd, buf, sizeof(buf), 0);
struct nlmsgerr *err = NLMSG_DATA((struct nlmsghdr *)buf);
if (err->error != 0) { errno = -err->error; /* fallito */ }
```

veth_create() — struttura del messaggio RTM_NEWLINK

Creare una veth pair richiede attributi **annidati**: `IFLA_LINKINFO` contiene `IFLA_INFO_DATA`, che contiene `VETH_INFO_PEER`.

Listing 4: Struttura ad albero del messaggio — da container.c

```
nlmsghdr  (RTM_NEWLINK | NLM_F_CREATE)
  ifinfomsg
  IFLA_IFNAME       = "veth0"       /* nome lato host   */
  IFLA_LINKINFO
    IFLA_INFO_KIND  = "veth"        /* tipo di link     */
    IFLA_INFO_DATA
      VETH_INFO_PEER                /* descrive veth1   */
        ifinfomsg   (peer)
        IFLA_IFNAME = "veth1"       /* nome lato cont.  */
```

Gli attributi annidati vanno **chiusi** aggiornando `rta_len` di ogni livello *dopo* aver aggiunto i figli — altrimenti il kernel vede lunghezze errate e scarta il messaggio.

```
/* Chiusura bottom-up degli rtattr */
peer_rta->rta_len  = end - peer;
info_data->rta_len = end - idata;
linkinfo->rta_len  = end - linfo;
```

Verifica sull'host

```
ip link show veth0
# veth0@veth1: <UP>
#    inet 172.17.0.1/24

ip link show veth1
# (non trovato: e' nel
#  container ns)
```

veth_move_to_ns() e setup_container_net()

Spostare veth1 nel container

```
/* RTM_NEWLINK sull'iface esistente
   + IFLA_NET_NS_PID = child_pid
   => il kernel sposta l'iface
      nel net ns del processo */
nl_addattr(nlh, IFLA_IFNAME,
    VETH_CONT, ...);

uint32_t pid = child_pid;
nl_addattr(nlh, IFLA_NET_NS_PID,
    &pid, sizeof(pid));
```

Dopo questa chiamata `veth1` **sparisce** dall'host e appare dentro il container. Lo spostamento è atomico: non c'è finestra in cui l'iface non appartiene a nessun ns.

Dentro il container (child_fn)

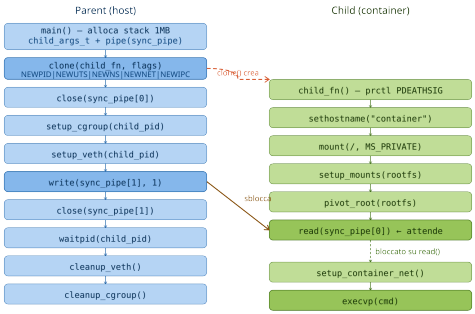
```
/* Eseguito dopo pivot_root,
   dentro il net ns isolato */
if_set_up("lo");
if_set_up(VETH_CONT);
if_add_addr(nl_fd,
    VETH_CONT,
    "172.17.0.2", 24);
```

Chiave: quando apriamo un socket `AF_NETLINK` qui dentro, il kernel lo vede nel *nostro* net ns — quindi `if_add_addr` agisce solo su `veth1`, non su `veth0` dell'host.

Verifica dall'interno del container

```
ip addr show          # veth1: 172.17.0.2/24, lo: 127.0.0.1
ping 172.17.0.1       # raggiunge l'host -> OK
ping 8.8.8.8          # Internet (solo se ip_forward + NAT attivi)
```

Sequenza completa — dal processo host al container



Build e test del container

Prepara l'ambiente:

```
# 1. Compila
gcc -Wall -o container container.c

# 2. Scarica un rootfs Alpine minimale (~3.5 MB)
mkdir rootfs
curl -sL https://dl-cdn.alpinelinux.org/alpine/v3.19/releases/x86_64/\
alpine-minirootfs-3.19.0-x86_64.tar.gz | tar xz -C rootfs

# 3. Avvia il container
sudo ./container ./rootfs /bin/sh
```

Verifica l'isolamento dall'interno:

```
# Hostname isolato
hostname            # -> container

# Processi: vedi solo i tuoi
ps aux              # solo /bin/sh e ps

# Filesystem: radice Alpine, non host
ls /                # bin etc lib usr var ...
cat /etc/os-release # -> Alpine Linux

# Rete: solo loopback
ip link show        # 1: lo: <LOOPBACK>
```

Verifica cgroups e limiti

```
# Dall'host: verifica il cgroup creato
ls /sys/fs/cgroup/container_demo/
# cgroup.procs  memory.max  pids.max  ...

cat /sys/fs/cgroup/container_demo/memory.max
# 67108864   (64 MB)

cat /sys/fs/cgroup/container_demo/pids.max
# 32

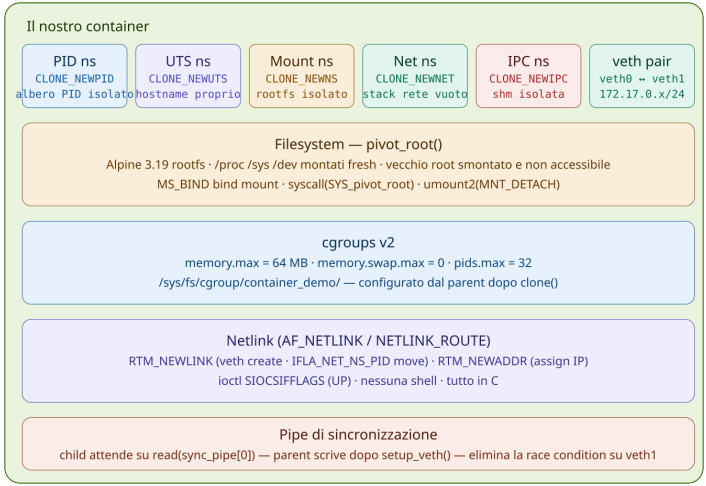
# PID host del container (es: 7842)
cat /sys/fs/cgroup/container_demo/cgroup.procs
# 7842

# Monitora memoria in tempo reale
watch cat /sys/fs/cgroup/container_demo/memory.current
```

Test fork bomb

```
# Dentro il container: pids.max=32 blocca la fork bomb
:(){ :|:& };:
# bash: fork: retry: Resource temporarily unavailable
```

Riepilogo — cosa abbiamo costruito



Cosa manca rispetto a Docker?

Implementato:

- ✓ PID, UTS, Mount, Net, IPC namespace
- ✓ cgroups v2 (mem, pids)
- ✓ `pivot_root()` sicuro
- ✓ Mount di /proc, /sys, /dev
- ✓ Alpine rootfs reale
- ✓ veth pair via Netlink (senza bridge)

Da aggiungere:

- ○ User namespace (rootless)
- ○ Bridge + multi-container
- ○ Overlay filesystem (layers)
- ○ Seccomp (syscall filtering)
- ○ Capabilities dropping
- ○ `/dev/pts` (terminali)
- ○ Cgroup namespace