# UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Ingegneria “Enzo Ferrari”

Corso di Laurea in Ingegneria Informatica


# IL PROCESSO DI INIT E LA SUA EVOLUZIONE NEI SISTEMI OPERATIVI GNU/LINUX


**Tutor:**
Chiar.mo Prof. Letizia Leonardi

**Elaborato di Laurea di:**
*Paolo Bruno*



**Anno accademico 2018-2019**

# Sommario

# Introduzione

Il presente elaborato sarà incentrato sul processo di inizializzazione dei sistemi operativi GNU/Linux e sarà rivolto sia ad utenti non esperti sia ad utenti che hanno già familiarità con questo ambito. Nello specifico, all'utente non esperto verranno presentate tutte le attività che si susseguono in questo stadio, mentre, all'utente esperto verranno esposte le motivazioni che hanno portato allo sviluppo di un nuovo progetto per l'init del sistema.

Prima di iniziare la trattazione è doveroso fare una precisazione sul concetto stesso di *inizializzazione*.

L'inizializzazione è la procedura con la quale si predispongono le condizioni iniziali appropriate per il corretto funzionamento di dispositivi hardware o programmi software.

Immaginiamo di dover imparare un nuovo linguaggio di programmazione, ad esempio il C, il nostro primo programma eseguibile, con molta probabilità, inizierà con istruzioni del tipo:

```
int a = 3;
int b = 4;

int somma = a + b;
```

Le prime istruzioni non fanno altro che dichiarare e inizializzare le variabili a e b.

Ora immaginiamo di aver acquistato una nuova stampante, colleghiamo il cavo di alimentazione e avviamo, sul display leggeremo qualcosa come "`inizializzazione in corso…`".

In entrambi i casi l'inizializzazione era fondamentale, senza di essa: nel primo caso avremmo ottenuto un valore della variabile somma non predicibile, mentre nel secondo caso non avremmo potuto stampare.

La stessa cosa accade all'avvio di un sistema operativo, esso diventa utilizzabile ed effettivamente operativo per l'utente finale solo dopo la fase di inizializzazione (init).

Con l'obiettivo di essere più chiari possibile, l'elaborato è organizzato in 7 capitoli.

Prima verrà fornita una panoramica di quello che avviene nelle fasi precedenti a quella di inizializzazione (capitolo 1).

Nei capitoli 2 e 3 sarà prima analizzato il vecchio processo di init, il **System Unix V init**; poi saranno trattate le motivazioni che hanno decretato il suo abbandono.

Il capitolo 4 introdurrà il sistema di init più diffuso attualmente, **systemd**, che risulta utilizzato su molte delle distribuzioni GNU/Linux esistenti, come Debian, Ubuntu, Fedora e Arch Linux.

Le migliorie introdotte da systemd saranno trattate nel capitolo 5.

Nel capitolo 6 verrà riportato il dibattito che si è sviluppato attorno alla domanda: "*quale sistema di init scegliere?*", prendendo come riferimento le vicende che hanno accompagnato lo sviluppo di Debian e Ubuntu.
L'ultima parte (capitolo 7) tratterà le numerose critiche nei confronti di systemd che, almeno in un primo momento, avevano acceso il dibattito e screditato la nuova suite software.

# 1 Processo di bootstrap

Il termine boot indica l'insieme di attività che vengono eseguite da un computer, dalla fase di accensione fino al caricamento completo, in memoria primaria, del kernel del sistema operativo, solitamente memorizzato in memoria secondaria.
Il termine originale inglese è bootstrap (fascetta di cuoio cucita sul bordo degli stivali per calzarli con più facilità), esiste un modo di dire inglese che rispecchia ciò che avviene nella fase di accensione: "*pull yourself up by your bootstra*ps" che può essere tradotto come "risolvi da solo i tuoi problemi, senza aspettare l'aiuto di altri" (Boot 2019).
L'analogia è legata al fatto che, in questa fase, il computer stesso esegue determinati processi in modo da portarsi in uno stato operativo che permetta l'esecuzione di ulteriori attività.
La presente trattazione, se non specificato diversamente, fa sempre riferimento al kernel Linux e a sistemi operativi Unix-like.
Vediamo cosa avviene nelle prime fasi (Jones 2006), senza scendere nei dettagli; la fase successiva al caricamento del kernel in memoria sarà analizzata nello specifico in seguito.

L'azione che mette in atto l'intero processo è svolta dal processore; quando questo riceve la giusta tensione di alimentazione legge la sua prima istruzione da un indirizzo di memoria fisso, in genere all'interno di una ROM; si tratta di un'operazione di salto incondizionato che sposta l'esecuzione al vero programma da chiamare in causa, il BIOS (Basic Input-Output System).
Il termine BIOS identifica sia l'insieme di routine software che gestiscono le periferiche del computer, sia la parte hardware del sistema, generalmente una memoria non volatile; in entrambi i casi non vi è correlazione tra quest'ultimo e un sistema operativo particolare.
Il BIOS infatti precede il caricamento di diversi OS: Windows, MacOS, sistemi Unix-like.
Il mercato attuale sta vedendo il passaggio dalla suite BIOS alla suite UEFI (Unified Estensible Firmware Interface) che principalmente è in grado di garantire migliori servizi di crittografia, di gestione energetica e di diagnostica (UEFI - wikipedia).
La presente trattazione non prenderà in considerazione il nuovo sistema UEFI nel descrivere le prime fasi di boot; dettagli su queste componenti esulano dallo scopo del nostro elaborato.

La funzione cardine del BIOS è il POST (Power On Self Test) per verificare il corretto funzionamento delle componenti fisiche.

Il POST è un auto diagnosi del computer e se la diagnosi evidenzia dei problemi il boot viene interrotto.

Le principali attività svolte sono:

- Verifica del codice stesso del BIOS;
- Individuazione e verifica della memoria primaria;
- Catalogazione di tutti i device e bus presenti;
- Selezione dei device utili per la continuazione della fase di bootstrap.

La presenza di errori viene manifestata, anche in assenza di display, con una sequenza di segnali acustici ("beep") che indicano periferica guasta e tipo di problema.

In seguito, vengono individuati, all'interno dei device utili, i settori di avvio che contengono un boot record valido.

Il BIOS esegue il primo boot record, in genere localizzato nel MBR (Master Boot Record) composto dai primi 512 byte del disco.

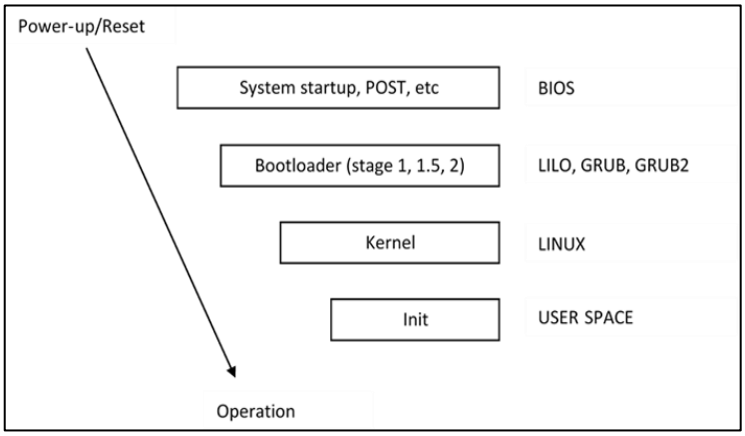



***Figura 1.1, Panoramica delle fasi di boot***

Il BIOS è solo una delle componenti interessate dal boot, una vista generale è presente nella ***Figura 1.1***.

Nella fase successiva interviene il **boot loader**, programma che durante la fase di avvio carica il kernel dalla memoria secondaria a quella primaria.

Il processore può in questo modo eseguire il codice del kernel; al momento GRUB, GRUB2 e LILO sono i più diffusi boot loader.

Consideriamo nei passaggi seguenti l'utilizzo di GRUB2 (GRand Unified Bootloader, versione 2).

Un generico MBR, per la dimensione cosi ridotta, presenta solo la parte di codice del boot loader relativa alla prima fase.

I 3 stadi, seguendo convenzionalmente la notazione del precedente GRUB1, sono i seguenti (Both 2017):

- Stage 1, ha lo scopo di localizzare ed avviare la parte di codice relativa allo stadio 1.5;

  Il file dello stage 1 prende il nome di boot.img.

- Stage 1.5, è contenuto nello spazio fra il boot record e la prima partizione del disco fisso, il file prende il nome di core.img.
  La dimensione è nettamente maggiore del record precedente.
  La funzione di questo record è di eseguire i drivers necessari per lo stage 2, principalmente quelli legati al riconoscimento ed utilizzo dei più comuni file system (EXT, FAT, NTFS).
- Stage 2, i file relativi ad esso sono nel percorso /boot, accessibile attraverso i driver caricati in precedenza.
  Viene prima caricato il kernel Linux in RAM e poi viene passato, a quest'ultimo, il controllo del sistema. Anche i file del kernel sono nella directory /boot insieme ad un'immagine della memoria centrale (*initrd*) e alla mappa dei dispositivi rigidi.

Il kernel, presente in memoria in formato compresso, può ora "auto-estrarsi"; come altri normali programmi utente esso è un ELF (Executable and Linkable Format) binario.
I software utente però, prima di essere caricati, devono disporre di alcune risorse (ad esempio memoria heap, memoria stack, descrittori di file) fornite generalmente dalla libreria standard *glibc*[1].
Per il caricamento del nucleo del sistema non può avvenire questo passaggio base.
Funzioni per la creazione esplicita dello stack e per la decompressione dei file ELF sono scritte in assembly, file *arc/x86/kernel/head_64S* nel codice sorgente del kernel (Chaiken 2018).
Al termine delle operazioni precedenti viene chiamata la funzione *start_kernel()*, presente in */init/main.c* che, a differenza delle precedenti, risulta indipendente da architetture specifiche.
Viene registrata la prima cpu, viene inizializzato lo scheduler e le strutture dati globali fondamentali al sistema oltre ai timer e agli handlers interrupt.
Questa fase è interamente sincrona, una procedura inizia solo al termine della precedente; questo fa sì che il registro di eventi visualizzabile con il comando *dmesg* sia strettamente affidabile.
L'output di *dmesg* è legato principalmente ai comandi presenti nella funzione *start_kernel()*.

[1] *Glibc* è la libreria standard del C del progetto GNU, viene utilizzata in diversi sistemi operativi, come quelli GNU/Linux su architettura x86.

In seguito, viene lanciata la procedura *rest_init()* che crea un thread per *kernel_init()* ed un secondo thread per *cpu_idle()*.

*Kernel_init()* continua le operazioni di inizializzazione del kernel: a titolo di esempio ricordiamo che in questa fase lancia la funzione SMP (Symmetric Multiprocessing) con *smp_init()* per preparare il sistema ad un funzionamento multicore.

Arrivati a questo punto, *kernel_init()* cerca un file che esegua il processo init per conto suo, vedere ***Figura 1.2***, il processo può essere costituito dal più moderno software **systemd**, dal predecessore **SysVinit** o da altre varianti che generalmente rispecchiano il funzionamento dei precedenti.

```
        /*
         * We try each of these until one succeeds.
         *
         * The Bourne shell can be used instead of init if we are
         * trying to recover a really broken machine.
         */
        if (execute_command) {
                ret = run_init_process(execute_command);
                if (!ret)
                        return 0;
                panic("Requested init %s failed (error %d).",
                      execute_command, ret);
        }
        if (!try_to_run_init_process("/sbin/init") ||
            !try_to_run_init_process("/etc/init") ||
            !try_to_run_init_process("/bin/init") ||
            !try_to_run_init_process("/bin/sh"))
                return 0;

        panic("No working init found.  Try passing init= option to kernel."
                "See Linux Documentation/admin-guide/init.rst for guidance.");
```

***Figura 1.2, Codice C della funzione kernel_init(), source code kernel Linux v5.0, /init/main.c linea 1113.***

Una volta configurati tutti i dispositivi, il kernel smonta l'immagine del disco initrd che ha permesso di caricare in memoria tutti i driver non compilati direttamente all'interno del kernel, crea un dispositivo di root, monta la partizione di root in sola lettura e libera la memoria utilizzata nella fase antecedente.

A questo punto il caricamento del kernel è compiuto, ma il computer non è ancora utilizzabile dall'utente finale dato che non ci sono altri processi in running; prima di eseguire qualsiasi applicativo il kernel commuta il processore da modalità reale a modalità protetta, in questo stato un programma utente non può accedere direttamente alla memoria ma deve chiamare i servizi del sistema operativo che gestiscono l'accesso ai dispositivi.

Il sistema, per continuare l'inizializzazione, passa il controllo al processo di init nominato in precedenza; per fare questo prima, verifica se sia stato richiesto un init particolare, passato sotto forma di variabile **execute_command** e prova ad eseguirlo, altrimenti prova a lanciare lo script di init cercandolo in 3 diverse locazioni.

Se il processo fallisce ulteriormente, prova ad avviare il processo di *Bourne shell* come descritto anche nel commento del codice in ***Figura 1.2****,* all'ennesimo fallimento il kernel Linux è costretto a lanciare un kernel panic.[2]

Solo quando init è in running il sistema inizia a comportarsi in modalità asincrona, preemptive e multiprocessore. Il processo di init avrà PID 1, il suo avvio indica che la fase di bootstrap è completa.

Segue la fase di startup.

[2] Un *kernel panic* è un'azione intrapresa da un sistema operativo Unix o Unix-like volta a identificare un errore fatale interno. L'equivalente sui sistemi Windows è la "Blue Screen of Death".

# 2 Processo di startup basato su System V init

Il processo di inizializzazione è generalmente localizzato in */sbin/init* ed è lanciato dal kernel nelle ultime fasi di boot.
Nella trattazione faremo riferimento a questo applicativo con i termini init, inizializzazione, processo di startup, usati in funzione di sinonimi.
Faremo prima riferimento al processo sysV init ormai superato da alternative più recenti come systemd.
Init è considerato il progenitore di tutti i processi, siano essi avviati automaticamente o per esplicita richiesta dell'utente.
Ha PID (Process ID) uguale a 1, essendo il primo processo eseguito dal kernel una volta caricato ed è anche il processo che prende "in affidamento" i processi orfani che terminato il padre, non avrebbero più un parent ID valido.
Il processo in questione avvia i servizi ed i programmi necessari a portare il sistema operativo in uno stato utile all'utente finale.

## 2.1 Funzionamento

Il funzionamento generale presenterà una serie di scripts e di locazioni di memoria che fanno riferimento a **System V Unix init program** (sysVinit) eseguito su sistema operativo GNU/linux Debian 7.0 Wheezy, kernel release 3.2.0 architettura amd64.
I sistemi operativi che nel tempo hanno usato il programma di boot sysVinit sono molti, generalmente viene seguito lo stesso procedimento descritto in questa sezione ma possono differire le locazioni delle cartelle e degli script.

Il processo di init (Negus 2012) per prima cosa legge il file di configurazione */etc/inittab* una riga alla volta.
Il file contiene l'elenco di quali script eseguire e in quali situazioni, la sintassi seguita dal file è *id:runlevel:azione:processo*.
La prima riga di codice riporta quale *runlevel* lanciare di default: Un runlevel è uno stato software del sistema e quando viene lanciato provoca l'interruzione di alcuni servizi e l'avvio di altri.
Le risorse del computer infatti possono essere utilizzate in modi differenti; ad esempio, un amministratore di rete potrebbe operare con il runlevel 1 che disabilità le funzioni di rete, mentre un utente normale potrebbe utilizzare il runlevel 3 che presenta tutti i servizi di base (compreso quelli di rete).

Ad esempio, la prima riga che si può trovare nel file */etc/inittab* potrebbe essere

**id:2:initdefault:**

che indica proprio l'informazione discussa precedentemente dove in questo caso il processo non è espresso perché *initdefault* definisce già completamente l'attività da svolgere (il runlevel da lanciare è il 2, specificato nel secondo campo).

I runlevel standard sono 7 (numerati da 0 a 6), ma solitamente possono essere usati solo quelli dal 2 a 5, i restanti hanno un utilizzo specifico per alcune situazioni del sistema operativo.

- Runlevel 0 (Halt): arresta tutti i servizi e ferma il sistema.
- Runlevel 1 (Single User Mode): modalità in cui solo l'account root è automaticamente loggato nel sistema mentre altri utenti non possono accedervi. I servizi di rete non sono avviati, è disponibile solo l'interprete a riga di comando; può essere utilizzato per la risoluzione di problemi software.
- Runlevel 6 (Reboot): riavvia il sistema e i suoi servizi.

Il file */etc/inittab* continua con le righe seguenti, analizzabili sempre secondo la sintassi sopra riportata:

**l0:0:wait:/etc/init.d/rc 0**

**l1:1:wait:/etc/init.d/rc 1**

**l2:2:wait:/etc/init.d/rc 2**

**l3:3:wait:/etc/init.d/rc 3**

**l4:4:wait:/etc/init.d/rc 4**

**l5:5:wait:/etc/init.d/rc 5**

**l6:6:wait:/etc/init.d/rc 6**

Chiaramente delle 7 righe verrà eseguita solo quella del runlevel di default specificato nella prima riga del file.

Per un runlevel *X*, viene invocata l'azione indicata, in questo caso "wait" cha prevede di avviare e poi attendere la terminazione del processo */etc/init.d/rc X (X* è appunto un numero da 0 a 6 passato come parametro alla funzione *rc)*.

Questo script si sposta nella cartella */etc/rcX.d* ed esegue gli script contenuti.

Un esempio di possibile contenuto della cartella rc2.d è:

**S10sysklogd S20anacron S20gpm S20makedev S50proftpd S99rmnologin S12kerneld S20apmd S20inetd S20postgresql S89cron S14ppp S20exim S20logoutd S20xfs S91apache**.

I nomi di questi file sono composti da 3 parti:

- S o K, indica se il processo in questione deve essere avviato (Start) o interrotto (Kill);
- Numero decimale, indica la priorità con cui questi devono essere avviati (in ordine ascendente);
- Nome del processo.

Quindi durante l'esecuzione di init viene letto il file */etc/inittab*, viene trovato il *runlevel* di default (ad esempio il runlevel 2), viene lanciato lo script */etc/init.d/rc 2* (2 come argomento), lo script si sposta nella cartella */etc/rc2.d* ed esegue gli script contenuti.

Notiamo che i file in */etc/rcX.d* sono in realtà dei link software che puntano a script in */etc/init.d*, in questo modo possono essere riusati facilmente in ogni runlevel che ne ha bisogno.

Nell'analisi del file inittab è stato tralasciato un dettaglio indipendente dai runlevel; soffermiamoci sulla riga **si::sysinit:/etc/init.d/rcS.**

Questa indica quale script eseguire per l'inizializzazione e configurazione del sistema durante il boot-time.

Lo script *rcS* si differenzia dai precedenti perché lancia ulteriori script che devono essere eseguiti per portare il sistema ad uno stato operativo.

Esso è eseguito sempre per primo, ad eccezione della modalità di avvio d'emergenza per la quale non viene eseguito.

È presente una sola riga di codice all'interno dello script:

**exec /etc/init.d/rc S** (S passato come parametro)

comando che a sua volta esegue tutti gli script contenuti in */etc/rcS.d* in ordine numerico/alfabetico.

Nel paragrafo seguente verranno approfonditi questi script.

## 2.2 Ulteriori script eseguiti per ogni runlevel

Gli script contenuti in */etc/rcS.d* sono eseguiti per l'inizializzazione e la configurazione del sistema; vengono eseguiti prima che init legga ed esegua gli script dettati dal runlevel di default (indicato nel file */etc/inittab*).

L'ordine di esecuzione è indicato dal numero progressivo che precede il nome, in caso di parità fa fede l'ordine alfabetico; viene riportata una lista dei principali script contenuti in */etc/rcS.d* considerando il sistema Debian descritto all'inizio della sezione 2.1, dato che i file potrebbero cambiare tra un SO e l'altro o anche tra versioni differenti dello stesso.
L'elenco presenta programmi demone, in inglese *daemon*, che si differenziano dal resto dei processi perché operano in background e tipicamente forniscono un servizio all'utente.
I demoni hanno nomi che convenzionalmente finiscono per "d", *httpd* ad esempio è il demone che gestisce il servizio http, *syslogd* è il demone che opera sui log di sistema e così via.

**S01mountkernfs.sh**
È il primo script che viene eseguito dopo che il processo di startup ha letto il file di configurazione.
La sua priorità è la più alta rispetto agli altri script da eseguire.
Monta i virtual file system (VFS) forniti dal Kernel e che sono necessari per le procedure successive.
Nella distribuzione analizzata vengono montati i file system seguenti:

- tmpfs su */run* e su */run/lock*; è un file system virtuale che poggia su memoria volatile, i file sono memorizzati temporaneamente e non sono scritti su disco rigido. All'avvio successivo tutti i dati andranno persi, un file system simile è il RAMDisk.
  L'utilizzo di tmpfs può avvenire anche per la partizione */tmp* (viene utilizzata la caratteristica del file system di poggiare su memoria volatile) e per */run/shm* (per la gestione della memoria condivisa[3]).
- proc file system su */proc*, è un ulteriore file system virtuale, in esso sono contenute informazioni sull'hardware e sui processi software in esecuzione.
  L'idea alla base di questa struttura è che in Linux qualsiasi informazione viene memorizzata sotto forma di file, proc contiene file virtuali che possono avere dimensione di zero byte ma riportare molte informazioni.
  I file */proc/meminfo* e */proc/filesystem* contengono informazioni rispettivamente sull'hardware e sulla configurazione del sistema; */proc/ide* contiene informazioni sui dispositivi fisici IDE.

[3] Le API POSIX per la memoria condivisa, che forniscono strumenti di comunicazione fra processi diversi, utilizzano locazioni di memoria /run/shm o /dev/shm.

L'obiettivo degli sviluppatori era quello di mantenere una certa organizzazione e consistenza fra la struttura delle directory e le macroaree del sistema in esecuzione.

- sysfs su */sys*; file system virtuale che ha un compito molto simile a proc; è infatti usato dal kernel per memorizzarvi informazioni riguardanti l'hardware poi sfruttate da altri programmi (come udevd).

  L'utilizzo di /sys rappresenta il tentativo di mantenere più pulito il file system proc, questo perché nel corso degli anni la struttura dati *device tree* che descrive tutte le componenti hardware del computer aveva appesantito eccessivamente procfs che contemporaneamente manteneva i dettagli dei processi in esecuzione.

**S02udevd**

È un gestore dispositivi per il kernel Linux, introdotto dalla versione 2.6 del kernel.

Viene eseguito come daemon (il processo lanciato è udevd) che principalmente si mette in ascolto degli eventi generati dal kernel quando un dispositivo viene collegato o rimosso.

Alla notifica di un evento, udevd può rispondere con una particolare azione definita nel suo database delle regole.

È eseguito in user space e gestisce l'intera cartella */dev* aggiungendo, rimuovendo o modificando i file relativi alle periferiche.

**S03mountdevsubfs.sh**

Script per il caricamento di ulteriori file system virtuali forniti dal kernel e che risiedono in */dev*.

**S04bootlogd**

Daemon non strettamente necessario per il corretto start-up del sistema, gestisce il programma di bootlog che cattura i messaggi di log in fase di boot.

In particolare, i suoi messaggi di log sono stati utilizzati per questo elaborato ai fini di verificare l'effettiva esecuzione dei programmi che concorrono alla fase di startup del sistema.

I messaggi, per impostazione predefinita, sono memorizzati nel file */var/log/boot*.

**S05keyboard-setup**

Configura la tastiera, in modo che siano riconosciuti correttamente i simboli ASCII.

La priorità è relativamente alta per permettere all'amministratore del sistema di interagire da tastiera, in caso di errori, nella fase di controllo dei file system che inizierà in seguito.

**S06hdparm**
Programma che consente la modifica e l'ottimizzazione di parametri dei dispositivi ATA/IDE (lettori cd/dvd, hard disk), in fase di avvio lo script può verificare il corretto funzionamento e registrare i parametri funzionali dei dispositivi in questione.

**S06hostname.sh**
Legge l'hostname della macchina in */etc/hostname* e aggiorna il parametro hostname del kernel.
Se */etc/hostname* è vuoto considera il valore corrente per hostname del kernel, se anche questo è nullo viene usato il valore "localhost".

**S06hwclock.sh**
Lo script gestisce una semplice operazione legata all'orologio di sistema.
L'orologio hardware del computer (RTC Real Time Clock) non possiede il concetto di tempo assoluto, inoltre tutte le operazioni per il settaggio dei formati e del fuso orario sono mantenute da un "orologio software" più flessibile e complesso.
I sistemi operativi Unix/Linux moderni salvano automaticamente l'ora di sistema nell'orologio hardware durante lo spegnimento e la recuperano dal RTC all'avvio.
Lo script hwclock.sh svolge proprio questa operazione.

**S07checkroot.sh**
Controlla il file system root e, in caso di errore, mostra importanti informazioni di log che permettono di individuare il problema.
Utilizza principalmente il comando di file system *consistency check (fsck)* per l'analisi e la correzione dei problemi più comuni. Errori fatali che non possono essere risolti dallo script automaticamente sono segnalati all'utente e richiedono un fix manuale come indicato dal segmento di codice in ***Figura 2.1*** che fa riferimento a questa situazione.
Lo script infatti dopo aver notato l'errore critico (variabile *rootfatal*), stampa i messaggi nei log di sistema ed infine avvia una shell per permettere all'amministratore stesso di risolvere la criticità.
Il commento che precede il codice riassume con semplicità che l'errore era irrisolvibile automaticamente e perché il controllo viene passato all'utente.

```
#
# Bother, said Pooh.
#
if [ "$rootfatal" = yes ]
then
        log_failure_msg "The device node $rootdev for the root filesystem is
                missing or incorrect or there is no entry for the root
                filesystem listed in /etc/fstab. The system is also unable to
                create a temporary node in /run.
                This means you have to fix the problem manually."
        log_warning_msg "A maintenance shell will now be started.
                CONTROL-D will terminate this shell and restart the system."
        # Start a single user shell on the console
        if ! sulogin $CONSOLE
```

***Figura 2.1, Parte dello script /etc/rcS.d/S07checkroot.sh su sistema Debian 7.0***

**S08checkroot-bootclean.sh**
Cancella i file system temporanei creati in fase di avvio dopo che il root file system è stato montato.

Lo script utilizza il comando *rm* (*remove*) prima di chiamare la funzione *clean_all()* definita in */lib/init/bootclean.sh* che rimuove i file system citati (ad esempio quelli montati in */run*, */run/lock* e */run/shm*).

**S08kmod**
Carica i moduli kernel dichiarati in */etc/modules.*

**S08mtab**
Ha lo scopo di aggiornare il file *mtab* per far sì che contenga una lista di tutti i file system montati e le informazioni ad essi correlati. Il contenuto deve essere coerente con lo stato del sistema; in particolare, il file *mtab* deve essere controllato perché potrebbe non presentare i nomi di quei file system virtuali montati in fase di boot.
Nelle prime fasi di avvio il file può risultare non scrivibile.

**S09 checkfs.sh**
La struttura e il funzionamento sono simile allo script S07; quindi controlla tutti i restanti file system e in caso di errore critico, avvia una shell su console; gli errori vengono stampati nei log (siano essi failure o warning).

**S10mountall.sh**
Monta tutti i file system locali descritti in *etc/fstab*; è in questa operazione che viene montato il file system ext4 (se definito del file */etc/fstab*).

Al termine dello script tutti i file system di cui il sistema operativo necessita sono stati montati.

**S11mountall-bootclean.sh**
Cancella i file system temporanei creati dopo che tutti i file system locali sono stati montati. Funzionamento simile a S08checkroot-bootclean.sh.

**S12procps**
Script che configura i parametri del kernel durante l'avvio.
I settaggi da impostare sono letti dal file */etc/sysctl.conf* e attraverso il comando *sysctl* vanno a sostituire i valori di default del kernel.

**S12urandom**
Salva e ripristina il *random seed*, utilizzato dal random number generator come punto di partenza per ottenere sequenze di numeri casuali, quando il SO viene acceso, spento o riavviato.
La generazione di numeri casuali è un compito particolarmente delicato per un sistema operativo; sono molti i servizi che possono richiedere numeri casuali (principalmente servizi legati alla crittografia e alla sicurezza informatica).
Per ottenere queste sequenze numeriche si considerano grandezze come i movimenti del mouse, i tempi di input della tastiera e altre misurazioni legate a specifici eventi non determinabili a priori.
Durante la fase di boot potrebbero non essere stati ancora caricati i driver di mouse e tastiera ed in generale i dispositivi di I/O attivi potrebbero essere pochi.
Per incrementare la casualità dei numeri generati anche in fase di avvio può essere fondamentale aver memorizzato un vecchio valore casuale precedentemente calcolato.
Attualmente molti processori implementano un random number generator in hardware.

**S14rcpbind**
Avvia/interrompe *rpcbind* daemon, servizio che converte i "program numbers" delle RPC (Remote Procedure Call[4]) nei numeri di porta del vecchio protocollo DARPA, oggi conosciuti come numeri di porta TCP/IP.

[4] Paradigma che consente ad un programma di eseguire subroutine o procedure su un computer diverso da quello sul quale è in esecuzione, in modo analogo ai normali processi locali.

**S16mountnfs.sh**
Script che si mette in attesa di quei processi che in background montano i file system di rete quando le interfacce vengono attivate.
Al termine dello script il sistema può sfruttare le interfacce di rete.

**S17mountnfs-bootclean.sh**
Pulisce i file system temporanei dopo che i file system di rete sono stati montati, funzionamento simile allo script S11[5].

**S18kbd e S19console-setup**
Script che impostano i corretti font, leggono la *charmap* di default e permettono un corretto riconoscimento dei caratteri UNICODE (nelle fasi precedenti, era garantito solo il riconoscimento dei caratteri ASCII).

**S20alsa-utilitis**
Script che memorizza e richiama le impostazioni riguardanti il driver ALSA (Advanced Linux Sound Architecture). ALSA è un framework che fornisce API per driver di schede audio.
Altri framework come PulseAudio utilizzano funzioni rese disponibili dalle API ALSA.

**S20x11-common**
Crea una directory in */tmp* e definisce le prime impostazioni necessarie all'installazione e al funzionamento di un sistema *X Window*; un gestore grafico che fornisce solo i componenti base per l'interfaccia grafica: lo spostamento delle finestre e l'interazione con le periferiche di input. Delega la gestione dell'interfaccia grafica come lo stile grafico delle applicazioni all'ambiente desktop installato sul computer in uso.

Gli script precedenti sono fondamentali per avere un sistema operativo utilizzabile, infatti molti di essi montano e analizzano i file system del computer oppure avviano servizi di basso livello.
La descrizione proposta nelle pagine precedenti estende quella presente all'interno degli script stessi.
Risultano indipendenti dal programma di init che li esegue ma devono essere adattati alle varie distribuzioni Linux.

[5] Il procedimento di montare un file system, analizzarlo e poi pulire i file temporanei non più richiesti avviene svariate volte, sia nella fase di startup che in quella di bootstrap.

# 3 Perché rimpiazzare Unix System V init

Il compito di un buon sistema di inizializzazione è di rendere operativo il computer e di farlo il più velocemente possibile.

Per un veloce ed efficiente avvio sono cruciali due cose: (Poettering, Rethinking PID 1 2010):

- avviare di meno
- avviare di più in parallelo

Esistono servizi che sicuramente saranno richiesti prima o poi (syslog o journal, D-Bus system bus) mentre ve ne sono altri che potrebbero non venire mai richiesti (bluetoothd quando non vi è un dispositivo bluetooth collegato, servizi di stampa se non vi è una stampante disponibile).

La seconda tipologia di servizi può non essere eseguita immediatamente in fase di boot per risparmiare risorse.

Avviare di più in parallelo significa eseguire più servizi nello stesso tempo per massimizzare l'uso di CPU e dell'IO bandwidth ed avere di conseguenza una riduzione dei tempi di avvio.

I sistemi precedenti a systemd nel tentativo di parallelizzare lo start-up senza creare conflitti mantenevano serializzata una parte significativa del flusso di lavoro (un servizio che aveva bisogno di D-Bus doveva attendere fino all'arrivo del segnale ready riferito a D-Bus prima di essere eseguito, inoltre entrambi questi processi richiedevano syslog che doveva essere inizializzato prima di qualsiasi altro daemon).

Il risultato era costi di sincronizzazione e serializzazione alti e un incremento di performance ridotto.

## 3.1 Nuove risorse da utilizzare: le socket

Se consideriamo dei tradizionali Unix daemon che dipendono dalle operazioni di altri servizi, notiamo come le comunicazioni tra i processi possono richiedere soltanto una risorsa: la creazione della **socket** con la quale task client e task server possono comunicare.

Riprendendo l'esempio tra parentesi del paragrafo precedente, i clienti di D-Bus attendono la socket */var/run/dbus/system_bus_socket* mentre i client di syslog attendono */dev/log* e così per tutte le dipendenze; non vi sono altre risorse da attendere.

L'idea alla base, non sfruttata dal vecchio init, è rendere queste socket disponibili subito e solo dopo eseguire i daemon necessari, passando la socket creata come parametro alla *exec()*.

L’operazione viene quindi divisa in due:

1) Creare le socket per tutti i daemon in un singolo step;
2) Lanciare i daemon;

Se ad esempio syslog e dei sui clienti vengono eseguiti insieme, in un certo momento può avvenire che il client mandi un messaggio nella socket */dev/log* quando syslog non è ancora completamente attivo.

Il client continuerà la sua attività (se la richiesta a syslog non blocca il proprio workflow) e quando il daemon syslog risulterà operativo, leggerà il messaggio dal socket buffer e risponderà alla richiesta.

In generale, la gestione delle dipendenze sarà più chiara e demandata alle socket, strutture gestite dal kernel; per i task che possono continuare la loro esecuzione senza attendere da subito la risorsa richiesta, le performance risulteranno migliorate.

Anche la stabilità del software viene influenzata positivamente, se i vari servizi possono risultare non disponibili (ad esempio a causa di un crash) le relative socket risultano comunque attive ed in grado di ricevere messaggi; quindi le richieste inviate verso un servizio bloccato saranno memorizzate nel buffer e lette successivamente dal servizio (che nel frattempo sarà stato riavviato); il sistema generale non risentirà del problema di un particolare processo.

Il primo init ad utilizzare questa caratteristica è *launchd* nei sistemi Apple, proprio questa particolarità progettuale era alla base della velocità di boot-up che MacOS poteva vantare rispetto ai suoi competitor.

## 3.2 Shell script obsoleti

**Lennart Poettering,** sviluppatore di systemd, in un articolo sul suo blog scriveva: *“shell scripts are evil. Shell is fast and shell is slow. It is fast to hack, but slow in execution”*, (Poettering, Rethinking PID 1 2010).

Gli script sono estremamente più lenti di un programma scritto in C (o in altri linguaggi di programmazione compilati) e secondo Poettering sono anche più fragili dato che il loro comportamento può cambiare da un ambiente all’altro rendendo il controllo più difficile.

Gli script shell devono essere interpretati da un *command-line interpreter* (ne esistono diversi) e nel corso degli anni si sono sviluppati “diversi dialetti” tutti considerabili come linguaggi di scripting.

In sintesi, Poettering sosteneva che il lavoro fatto da molti script poteva essere eseguito da un programma scritto in C oppure poteva essere spostato internamente al sistema di init.

L'articolo a sostegno della tesi suggeriva*: "A good metric for measuring shell script infestation of the boot process is the PID number of the first process you can start after the system is fully booted up. Boot up, log in, open a terminal, and* type *echo $$. Try that on your Linux system, and then compare the result with MacOS! (Hint, it's something like this: Linux PID 1823; MacOS PID 154, measured on test systems we own.)".*

Il comando **echo $$** stampa a video il Process IDentifier (PID) del processo che sta eseguendo la shell corrente.

Il PID è un numero intero che il sistema operativo assegna ad ogni processo in esecuzione per identificarlo in maniera univoca.

I criteri per l'assegnazione del process ID seguono tradizionalmente la logica di conferire al nuovo task il più piccolo valore che non sia correntemente assegnato ad un qualche altro processo.

L'obiettivo era di ottenere una stima del numero dei processi attivi durante il boot up, calcolabili come **valore PID del processo terminale – 1**;

Riproducendo "l'esperimento" descritto (i risultati vogliono essere solo indicativi e non sono da considerare completi) notiamo che:

- PID ottenuto su Debian 7.0, kernel release 3.2.0 architettura amd64: **3365**; la configurazione utilizza **sysV init** come processo di start-up.
- PID ottenuto su Debian 10.0, kernel release 4.19.0 architettura amd64: **1251**; la configurazione utilizza il moderno **systemd**.

Nel primo caso il numero di processi (nati principalmente dall'esecuzione degli shell script) è nettamente maggiore del secondo; presupponendo che la velocità di startup sia fortemente correlata al numero di attività da eseguire in fase di avvio, la configurazione con systemd è nettamente più veloce.

System Unix V init, per come era stato progettato, era poco aggiornabile e scarsamente modulare[6]; difficilmente avrebbe potuto ricevere un qualche upgrade per il miglioramento delle performance. Queste considerazioni, insieme alle successive, portarono all'ideazione di nuovi sistemi di init e alla riscrittura di quegli script che impattavano maggiormente sui tempi di avvio.

[6] La logica del programma all'interno del codice sorgente risiedeva, quasi completamente, nel file init.c; il file sfiorava le 3000 righe di codice nella versione 2.88.

## 3.3 Tenere traccia dei processi con i control groups

Un altro aspetto che necessitava di essere rivisto era il modo di tener traccia dei processi.

In Linux, il processo con PID 1 ha sempre ricoperto un ruolo fondamentale, infatti è spesso definito come il progenitore comune di tutti i processi (particolarità mostrata in ***Figura 3.1***) e ha funzionalità che gli altri task non possiedono.

```
$pstree -g
systemd(1)-+-ModemManager(429)-+-{ModemManager}(429)
           |                   `-{ModemManager}(429)
           |-NetworkManager(435)-+-dhclient(487)
           |                     |-{NetworkManager}(435)
           |                     `-{NetworkManager}(435)
           |-accounts-daemon(436)-+-{accounts-daemon}(436)
           |                      `-{accounts-daemon}(436)
           |-alsactl(425)
           |-avahi-daemon(433)---avahi-daemon(433)
           |-boltd(1267)-+-{boltd}(1267)
           |             `-{boltd}(1267)
           |-colord(702)-+-{colord}(702)
           |             `-{colord}(702)
           |-cron(423)
           |-dbus-daemon(427)
           |-fwupd(1254)-+-{fwupd}(1254)
           |             |-{fwupd}(1254)
           |             |-{fwupd}(1254)
           |             `-{fwupd}(1254)
           |-gdm3(467)-+-gdm-session-wor(467)-+-gdm-wayland-ses(749)-+-
```

***Figura 3.1, Output del comando "pstree -g" eseguito da terminale su systema Debian 10.0***

Deve essere in grado di riavviare un processo che è stato interrotto, terminare un servizio di cui non si ha più bisogno, raccogliere informazioni in caso di crash, registrare i log di sistema utili all'amministratore e molte altre operazioni.

Alcune di queste, come interrompere completamente un servizio, sono difficili da mettere in pratica in alcune situazioni; il motivo principale è che tutti i processi in Linux possono chiamare la *fork()* un numero indefinito di volte e si può presentare il problema di avere processi figli che continuano anche dopo la terminazione del padre.

Gestire le varie gerarchie di processi che si vengono a creare in un sistema durante e dopo il boot può essere fatto sfruttando una risorsa fornita direttamente dal kernel e non utilizzata nel vecchio processo sysV init, i **control group** (cgroup).

I Control Groups permettono di creare una gerarchia di gruppi di processi, memorizzata direttamente in un file system virtuale e facilmente accessibile.

Il processo figlio, dopo che la *fork()* è stata chiamata, acquisisce lo stesso gruppo del padre; non può sfuggire a questa classificazione se non possiede i privilegi per accedere al file system virtuale.

L'utilizzo dei cgroup permette di individuare facilmente i processi legati ad un servizio e quindi di arrestarli.

Un buon sistema di monitoraggio dovrebbe permette anche di controllare l'ambiente di esecuzione dei vari task.
Infatti, sfruttando le funzionalità del kernel Linux è possibile settare limiti di utilizzo CPU, I/O, limiti di memoria, limiti di accesso; tutti riferibili a processi specifici o a interi cgroup.
La necessità di introdurre queste caratteristiche ovviamente non presenti in system V init si stava facendo sempre più pressante.

# 4 Processo di startup basato su systemd

Il processo systemd è lanciato dal kernel nelle ultime fasi di boot, rappresenta il successore del vecchio Unix System V init; è considerato il progenitore di tutti i successivi processi avviabili nell'host e come il precedente init ha PID 1.
Systemd (Software and Service Manager daemon) è nello specifico una suite di software che ha l'obiettivo di unificare la configurazione base delle principali distribuzioni Linux.
Lo sviluppo è iniziato da parte di **Lennart Poettering** e **Kay Sievers**, entrambi ingegneri alla Red Hat e la prima release risale a marzo 2010.
Le motivazioni che hanno spinto a rimpiazzare il vecchio processo saranno analizzate nel dettaglio dopo una visione sul funzionamento generale.
Il processo init di systemd è solo uno dei tanti task della suite avviati in fase di boot e arrestati in fase di shutdown.
Oggi all'interno della suite vi sono comandi che hanno superato le funzionalità di base, infatti il software può risultare responsabile del power management, della crittografia, della rete e di altre funzioni che hanno via via ampliato le potenzialità di systemd.
L'architettura così costituita viola, secondo parte della comunità del software libero, il principio Unix: *do one thing and do it well*.

## 4.1 Funzionamento generale

Il System Manager Bootup di systemd utilizza dei file di configurazione denominati **unit file** (anche riferibili con il termine unità) che sostituiscono i tradizionali script di shell.
Le unità rappresentano anche i tasks svolti da systemd dove i più comuni sono servizi, punti di montaggio, device, socket e timer identificati da file che terminano rispettivamente in .service, .mount, .device, .socket, .timer.
Un'unità è internamente divisa in sezioni identificate da "[]", ad esempio [Unit], [Install] che contengono le opzioni di configurazione e [Service] che presenta informazioni utili per l'avvio di servizi.
I tipi di unit file sono (Systemd 2019):

- .service
- .socket
- .device
- .mount
- .automount
- .swap

- .target
- .path
- .timer
- .snapshot
- .slice
- .scope

I file **.target** (target unit) sono usati per raggruppare unità e per indicare al sistema i punti di sincronizzazione noti del processo di start-up.

I due esempi presentati in ***Figura 4.1*** e ***4.2*** mostrano il contenuto rispettivamente di bluetooth.target e bluetooth.service e la loro suddivisione interna.

```
[Unit]
Description=Bluetooth
Documentation=man:systemd.special(7)
StopWhenUnneeded=yes
```

***Figura 4.1, Contenuto del file /lib/systemd/system/bluetooth.target***

```
[Unit]
Description=Bluetooth service
Documentation=man:bluetoothd(8)
ConditionPathIsDirectory=/sys/class/bluetooth

[Service]
Type=dbus
BusName=org.bluez
ExecStart=/usr/lib/bluetooth/bluetoothd
NotifyAccess=main
#WatchdogSec=10
#Restart=on-failure
CapabilityBoundingSet=CAP_NET_ADMIN CAP_NET_BIND_SERVICE
LimitNPROC=1
ProtectHome=true
ProtectSystem=full

[Install]
WantedBy=bluetooth.target
Alias=dbus-org.bluez.service
```

***Figura 4.2, Contenuto del file lib/systemd/system/bluetooth.service***

La ***Figura 4.2*** presenta anche la sezione [Service] con la riga ExecStart=*/urs/lib/bluetooth/bluetoothd* che indica il processo da avviare per mettere in atto il servizio Bluetooth.

All'interno dei file target sono presenti informazioni che rimandano alla documentazione (systemd manpages s.d.) e che aiutano a comprendere la catena di dipendenze che si crea in fase di boot.

Il primo target di cui vengono considerate le dipendenze è il default.target che generalmente è un link software al file graphical.target, entrambi localizzati nella cartella /lib/systemd/system.

```
[Unit]
Description=Graphical Interface
Documentation=man:systemd.special(7)
Requires=multi-user.target
Wants=display-manager.service
Conflicts=rescue.service rescue.target
After=multi-user.target rescue.service rescue.target display-manager.service
```

***Figura 4.3, Contenuto del file /lib/systemd/system/graphical.target con dichiarazione delle dipendenze***

Nella ***Figura 4.3*** viene riportato l'ultimo target ad essere considerato in fase di startup il quale specifica che devono essere caricati prima i servizi di livello inferiore.
Al suo interno, possiamo ritrovare altre righe utili come "**Requires**=", "**Wants**=" e "**After**=" oltre a riconoscere la presenza di una singola sezione [Unit].

- "Requires=", indica che la unit corrente richiede le altre unità elencate dopo "=";
- "Wants=", indica che al lancio dell'unità corrente, in genere un servizio, vengono lanciate anche le unità elencate; il corretto avvio di queste non influenzerà l'unità attuale.
- "After=", indica che la unit verrà eseguita solo dopo le unità elencate dopo "=";

Segue, risalendo l'albero delle dipendenze, il multi-user.target che sincronizza servizi che impostano l'ambiente di sistema per il supporto della multiutenza; quest'ultimo target presenta sotto-unità da eseguire in */etc/systemd/system/multi-user.target.wants*.
Il tentativo di aumentare la comprensione è evidente anche nella scelta dei nomi dei file.
Il target da cui dipendono i precedenti è basic.target che gestisce vari servizi, in particolare il gestore grafico; può fare riferimento alla directory */etc/systemd/system/basic.target.wants*.
Ricordiamo che un file .target è solo il modo per organizzare le dipendenze e sincronizzare la fasi di boot; non è uno script e non esegue in prima persona comandi ma si fa riferimento ad esso perché è un utile marcatore per descrivere le fasi di startup.
Basic.target dipende da Sysinit.target che sincronizza servizi di sistema fondamentali come il montaggio del file system, lo spazio di swap e servizi legati a device connessi al computer.
Questo target, come i precedenti, è raggiunto quando tutte le unità da cui dipende sono state caricate.
L'ultima dipendenza ha il nome local.fs.target che gestisce solo servizi di livello inferiore, non legati all'utente; in questa fase vengono richiamati processi che lavorano sul contenuto dei file */etc/fstab* e */etc/inittab* incontrati precedentemente.

Durante l'esecuzione di systemd vengono attivate tutte le unità che sono dipendenze del file default.target; dalla documentazione ufficiale presentiamo un grafico (***Figura 4.4***) che mostra l'ordine e la logica generale delle unità che prendono parte al processo di init.
Consideriamo le unità e i relativi servizi che devono essere completate prima di raggiungere sysinit.target: il montaggio dei file system, l'impostazione dei file di swap, l'avvio di udev, l'impostazione del random generator seed, l'esecuzione di servizi di crittografia, varie API di basso livello.
Le attività da svolgere in questa parte del processo sono molte ma è importante ricordare che, a differenza del predecessore sysV init, systemd sfrutta processi paralleli, più efficienti ed in grado di sfruttare al meglio i processori più recenti; questa caratteristica emerge già nella ***Figura 4.4***. Le differenze tra i due sistemi saranno analizzate in seguito.

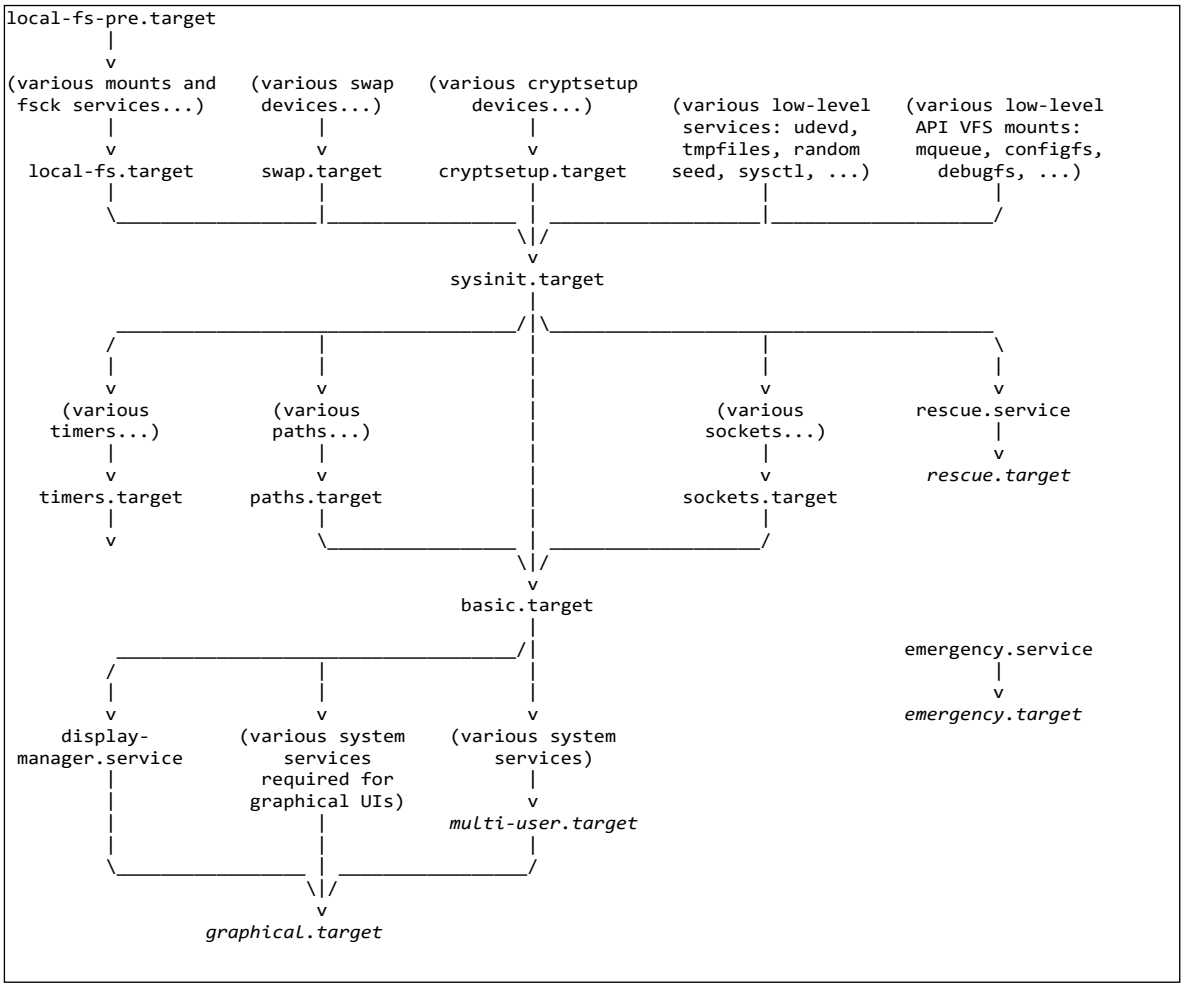



***Figura 4.4, Grafico sulle unit eseguite durante il bootup (pagina di "bootup" del manuale di sistema)***

## 4.2 Principali funzionalità introdotte

Abbiamo descritto interamente l'avvio governato da systemd, ma è importante ricordare che questo daemon non è un semplice programma eseguito solo per il bootup dello spazio utente; è una suite software molto più ampia.

È evoluta nel tempo, includendo nuovi binari e racchiudendo al suo interno nuove funzionalità; i componenti *core* sono i seguenti (Systemd 2019):

- **systemd**, System e Service Manager già considerato in precedenza;
- **systemctl**, per controllare lo stato del gestore di sistema e dei servizi;
- **systemd-analyze**, per verificare le performance della fase di boot e tracciare le informazioni riguardanti le unità che intervengono nel processo.

Utilizzeremo i comandi di *systemctl* e *systemd-analyze* per considerare alcuni dettagli della fase di boot del computer.

Il comando **systemd-analyze**, seguito dal parametro **time**, stampa a video il tempo impiegato per il caricamento del kernel e per il caricamento dello spazio utente; nell'esempio riportato in ***Figura 4.5*** il checkpoint rappresentato da graphical.target che principalmente sincronizza i servizi necessari per l'ambiente grafico viene raggiunto dopo 39.7s.

```
$ systemd-analyze time
Startup finished in 3.916s (kernel) + 39.785s (userspace) = 43.702s
graphical.target reached after 39.733s in userspace
```

***Figura 4.5, Output del comando "systemd-analyze time" eseguito da terminale***

Un ulteriore comando, utile per analizzare e migliorare i tempi di boot è **systemd-analyze blame.**

L'output, ***Figura 4.6***, è una lista di servizi e rispettivi tempi di avvio, ordinata dal servizio più "lento" (servizio che impiega più tempo nella fase di avvio) a quello più rapido.

```
$ systemd-analyze blame
         30.811s plymouth-quit-wait.service
         16.026s man-db.service
         12.617s apt-daily.service
         10.808s udisks2.service
          5.327s ModemManager.service
          3.705s dev-sda1.device
          3.611s accounts-daemon.service
          3.043s NetworkManager.service
          2.892s avahi-daemon.service
          2.887s logrotate.service
          2.818s switcheroo-control.service
          2.810s pppd-dns.service
          2.803s systemd-logind.service
          2.802s wpa_supplicant.service
          2.727s alsa-restore.service
          2.727s rsyslog.service
          2.110s ssh.service
          1.576s systemd-udevd.service
          1.253s apparmor.service
          1.215s systemd-tmpfiles-clean.service
          1.128s gdm.service
          1.064s fwupd.service
           679ms polkit.service
```

***Figura 4.6, Output del comando "systemd-analyze blame" eseguito da terminale***

Altro comando estremamente utile per l'analisi è **systemd-analyze plote**.

L'output di questo comando è un file .svg che mostra graficamente l'attivazione di tutte le **units**.

La concorrenza dei processi infatti porta con sé un problema "secondario": come visualizzare le informazioni nei log.

Questi sono particolarmente adatti a mostrare quello che avviene durante le varie fasi di un processo in serie ma possono dimostrarsi inadatti quando il tutto avviene in modo parallelo (con punti di sincronizzazione e con molti processi avviati nello stesso istante).

Un esempio di file .svg è mostrato nella ***Figura 4.7***.


***Figura 4.7, File .svg generato dal comando "systemd-analyze plot"***

Il grafico evidenzia come il kernel impieghi circa 4s per diventare operativo, poi, come spiegato precedentemente, la cpu viene passata a systemd.

A 6 secondi dall'avvio inizia il caricamento di tutti i daemon e servizi, molti dei quali avviati in parallelo; in questo caso l'analisi del file .svg è molto più semplice e immediata dell'analisi dei logs.

I comandi systemctl sono generalmente usati per controllare lo stato del System e Service Manager daemon.

Le funzioni interne di questo comando sono molte; ci soffermiamo su **systemctl list-units** per la stampa di tutte le unit caricate in memoria e che risultano attive, in attesa o che mostrano un errore di esecuzione, vedere ***Figura 4.8***.

```
UNIT                            LOAD   ACTIVE SUB       DESCRIPTION
proc-sys-fs-binfmt_misc.automount loaded active waiting   Arbitrary Executable File
sys-module-fuse.device          loaded active plugged   /sys/module/fuse
sys-subsystem-net-devices-enp0s3.device loaded active plugged   82540EM Gigabit
Ethernet
-.mount                         loaded active mounted   /
dev-hugepages.mount             loaded active mounted   Huge Pages File System
dev-mqueue.mount                loaded active mounted   POSIX Message Queue File
run-user-1000-gvfs.mount        loaded active mounted   /run/user/1000/gvfs
run-user-1000.mount             loaded active mounted   /run/user/1000
sys-kernel-debug.mount          loaded active mounted   Kernel Debug File System
systemd-ask-password-plymouth.path loaded active waiting   Forward Password Requests
systemd-ask-password-wall.path loaded active waiting   Forward Password Requests
init.scope                      loaded active running   System and Service Manage
```

***Figura 4.8, Output non completo del comando "systemctl list-units"***

Le unit che sono andate incontro a *failure* durante il caricamento sono elencate dal comando **systemctl –failed**.

Le funzioni mostrate precedentemente non sono un elenco completo di tutte quelle rese effettivamente disponibili dal software systemd; il breve scopo di questa sezione era di evidenziare come la nuova “suite di init” sia in grado di accompagnare l’amministratore di sistema nell’analisi e nella correzione di eventuali problemi in fase di avvio, nonché nel miglioramento delle performance durante il caricamento dei molti servizi.

# 5 Vantaggi introdotti dal nuovo systemd

Abbiamo già visto, nel capitolo 3, alcune delle mancanze di sysV init che hanno motivato il passaggio ad una suite software più moderna per la gestione del processo di inizializzazione. Inoltre, nel capitolo 4 abbiamo analizzato il funzionamento di systemd e le caratteristiche introdotte da esso.

La lettura di queste sezioni permette già di cogliere molti dei vantaggi introdotti da systemd; in questo capitolo si vuole fornire una panoramica completa dei vantaggi che il nuovo daemon ha portato con sé.

I possibili successori del vecchio init erano molti, **upstart**, **runit**, **procd**, **initng** solo per citarne alcuni, questi però sono stati usati solo in determinati sistemi operativi e per periodi di tempo limitati; i pro introdotti da systemd (Poettering, systemd for administrator 2012) erano tali da renderlo il più diffuso dopo solo due anni dal suo rilascio.

## 5.1 Ottimizzazione delle performance

La caratteristica principale di systemd è quella di aver ridotto i tempi di inizializzazione.

Il software, come emerso più volte, è in grado di sfruttare il parallelismo fra processi.

La ***Figura 4.4*** (sezione 4.1 Funzionamento generale) mostra graficamente l'ordine con la quale i processi vengono eseguiti, alcuni sono affiancati da altri e vengono elaborati in maniera concorrente.

Il comportamento di sysV init era diverso, se concentriamo la nostra attenzione alla semplice fase di lettura del file */etc/inittab* abbiamo presentato righe del tipo:

**l0:0:wait:/etc/init.d/rc 2**

il comando *wait* indica a init di eseguire lo script **rc 2** e poi **attendere** la sua terminazione.

L'approccio che emerge in questo caso e in tutti gli altri script del paragrafo 2.2 è estremamente diverso rispetto al nuovo init; infatti si può notare come tutti i passaggi della procedura di boot erano avviati in serie, solo quando un'attività era completamente terminata, si poteva passare alla successiva con priorità minore.

Le socket come descritte nel capitolo 3 sono lo strumento cardine che ha permesso di gestire in maniera corretta il parallelismo e le dipendenze.

## 5.2 Processo di boot semplice (The Self-Explanatory Boot)

L'esecuzione di più unità nello stesso istante insieme all'albero delle dipendenze gestito da systemd può rendere più arduo comprendere il processo di boot.

Questa affermazione non è sempre vera dato che per molti aspetti il nuovo init risulta più facile da interpretare del predecessore.

Abbandonando l'utilizzo intensivo di shell script si rende indirettamente più morbida la curva di apprendimento; infatti in passato, per un'analisi dei task di init, occorreva la conoscenza approfondita del linguaggio di scripting **Bourne Shell**.

Systemd viene anche accompagnato da centinaia di pagine di manuale, l'elaborato stesso utilizza come fonte il manuale di sistema.

L'accesso alla documentazione del software viene velocizzato e semplificato grazie al nuovo **self-explanatory boot process** secondo cui ogni unit in systemd deve ora includere un riferimento alla sua documentazione, in modo che l'utente possa subito capire lo scopo dell'unità, il contesto e come configurarla.

Attraverso il comando **systemctl status** (***Figura 5.1***) si ottengono tutte le informazioni relative ad una unit, nella sezione "Docs" dell'output ottenuto è presente il collegamento alle pagine di documentazione.

```
$ systemctl status systemd-logind.service
systemd-logind.service - Login Service
         Loaded: loaded (/usr/lib/systemd/system/systemd-logind.service; static)
         Active: active (running) since Mon, 25 Jun 2012 22:39:24 +0200; 1 day and 18h ago
           Docs: man:systemd-logind.service(7)
                 man:logind.conf(5)
                 http://www.freedesktop.org/wiki/Software/systemd/multiseat
       Main PID: 562 (systemd-logind)
         CGroup: name=systemd:/system/systemd-logind.service
                 └ 562 /usr/lib/systemd/systemd-logind

Jun 25 22:39:24 epsilon systemd-logind[562]: Watching system buttons on /dev/input/event2 (Power Button)
Jun 25 22:39:24 epsilon systemd-logind[562]: Watching system buttons on /dev/input/event6 (Video Bus)
Jun 25 22:39:24 epsilon systemd-logind[562]: Watching system buttons on /dev/input/event0 (Lid Switch)
Jun 25 22:39:24 epsilon systemd-logind[562]: Watching system buttons on /dev/input/event1 (Sleep Button)
Jun 25 22:39:24 epsilon systemd-logind[562]: Watching system buttons on /dev/input/event7 (ThinkPad Extra
Buttons)
Jun 25 22:39:25 epsilon systemd-logind[562]: New session 1 of user gdm.
Jun 25 22:39:25 epsilon systemd-logind[562]: Linked /tmp/.X11-unix/X0 to /run/user/42/X11-display.
Jun 25 22:39:32 epsilon systemd-logind[562]: New session 2 of user lennart.
Jun 25 22:39:32 epsilon systemd-logind[562]: Linked /tmp/.X11-unix/X0 to /run/user/500/X11-display.
Jun 25 22:39:54 epsilon systemd-logind[562]: Removed session 1.
```

***Figura 5.1, Output del comando "systemctl status" riferito al servizio systemd-logind***

## 5.3 Tracking dei processi avanzato

Systemd introduce un controllo sui processi basato su cgroup il cui principio di funzionamento è stato trattato nel capitolo 3.

I servizi più importanti all'interno di un sistema Linux hanno la caratteristica comune di mantenere molti processi distinti a lavoro, pensiamo ad esempio a un daemon che si mette in ascolto e per ogni richiesta ricevuta genera un processo figlio in grado di eseguirla.
L'output di comandi come **ps** risulterà subito intasato da processi difficili da riconoscere e che sono riferibili al servizio che svolgono solo per il loro nome (proprietà che i processi possono cambiare liberamente).
Questo motivo, insieme a considerazioni su come limitare l'uso di risorse condivise, ha portato gli sviluppatori del nuovo init a sfruttare i control groups.
Viene riportato un esempio che chiarisce meglio la potenzialità di questi raggruppamenti.
Eseguendo il comando **ps -e** otteniamo la stampa in ***Figura 5.2***, notiamo che i processi con pid 439 e 462 si riferiscono al daemon del servizio **Avahi**[7] che automatizza la configurazione della rete locale quando nuove periferiche si interconnettono ad essa.
Solo il nome evidenzia che i due processi sono correlati fra loro e con molta probabilità si scambiano informazioni per il corretto funzionamento del servizio.
Se per qualche motivo avessero avuto un nome meno esplicito l'amministratore avrebbe fatto fatica a comprendere l'utilità di questi processi e il loro legame.

```
  PID TTY          TIME CMD
    1 ?        00:00:01 systemd
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 rcu_gp
   32 ?        00:00:00 ksmd
   60 ?        00:00:00 kthrotld
   61 ?        00:00:00 ipv6_addrconf
   71 ?        00:00:00 kstrp
  108 ?        00:00:00 ata_sff
  279 ?        00:00:00 systemd-timesyn
  432 ?        00:00:00 accounts-daemon
  439 ?        00:00:00 avahi-daemon
  440 ?        00:00:00 wpa_supplicant
  441 ?        00:00:00 cron
  443 ?        00:00:00 rsyslogd
  462 ?        00:00:00 avahi-daemon
  477 ?        00:00:00 gdm3
  491 ?        00:00:00 sshd
```

***Figura 5.2, Output del comando "ps -e" eseguito da terminale***

Ora analizziamo l'output (nella ***Figura 5.3***) di un altro comando, **systemd-cgls**, esso mostra gli stessi processi ma questa volta raggruppa i task considerando i cgroup di cui fanno parte.
In entrambi i comandi l'output è stato tagliato per una migliore formattazione della pagina.

[7] Il servizio Avahi ad esempio permette, ad un computer che si è appena collegato alla rete locale, di individuare stampanti condivise o altri dispositivi nella rete senza la necessità di una configurazione manuale.

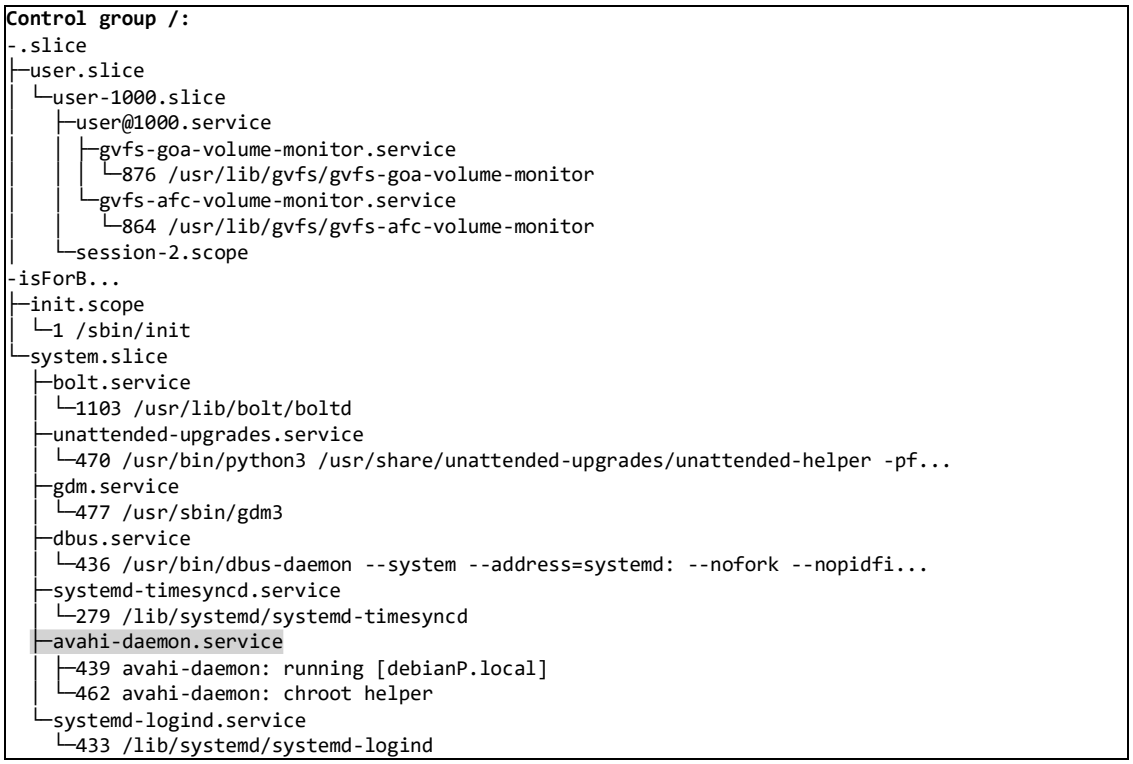



***Figura 5.3, Output del comando "systemd-cgls" eseguito da terminale***

La visualizzazione precedente evidenzia come i processi con pid 439 e 462 fanno riferimento ad un servizio preciso: il servizio Avahi; per semplicità è stato considerato un caso in cui i task hanno nomi espliciti e sono solamente due ma in contesti in cui intervengono molti processi tutti riferibili ad un servizio, l'uso di cgroup e le nuove funzionalità di systemd sono fondamentali per l'amministratore.

## 5.4 Gestione delle dipendenze migliorata

Il modo con il quale vengono gestite le dipendenze tra i vari servizi cambia dal vecchio init al nuovo.

Un confronto tra il funzionamento di System Unix V init e systemd permette di notare subito la differenza.

La gestione delle dipendenze, svolta dal predecessore di systemd, può risultare eccessivamente scarna paragonata al nuovo init ma consideriamo meglio l'ambito di utilizzo dei diversi software: sysVinit era destinato a sistemi statici, configurati una prima volta e aggiornati solo in caso di necessità dall'admin di sistema; systemd è stato costruito per adattarsi meglio ai sistemi dinamici moderni, questi ultimi presentano ad esempio componenti che possono essere collegati, scollegati, installati, disinstallati o momentaneamente disattivati.

Il metodo rudimentale usato da sysV init per gestire le dipendenze era quello di eseguire gli script necessari in ordine ascendente rispetto al numero che precedeva il nome dello script.
La ***Figura 5.4*** mostra gli script contenuti in */etc/rcS.d* elencati secondo l'ordine di avvio.

```
S01mountkernfs.sh -> ../init.d/mountkernfs.sh
S02udev -> ../init.d/udev
S03mountdevsubfs.sh -> ../init.d/mountdevsubfs.sh
S04bootlogd -> ../init.d/bootlogd
S05keyboard-setup -> ../init.d/keyboard-setup
S06hdparm -> ../init.d/hdparm
S06hostname.sh -> ../init.d/hostname.sh
S06hwclock.sh -> ../init.d/hwclock.sh
S07checkroot.sh -> ../init.d/checkroot.sh
S08checkroot-bootclean.sh -> ../init.d/checkroot-bootclean.sh
S08kmod -> ../init.d/kmod
S08mtab.sh -> ../init.d/mtab.sh
S09checkfs.sh -> ../init.d/checkfs.sh
S10mountall.sh -> ../init.d/mountall.sh
S11mountall-bootclean.sh -> ../init.d/mountall-bootclean.sh
S12pppd-dns -> ../init.d/pppd-dns
S12procps -> ../init.d/procps
S12udev-mtab -> ../init.d/udev-mtab
S12urandom -> ../init.d/urandom
S13networking -> ../init.d/networking
S14rpcbind -> ../init.d/rpcbind
S15nfs-common -> ../init.d/nfs-common
S16mountnfs.sh -> ../init.d/mountnfs.sh
S17mountnfs-bootclean.sh -> ../init.d/mountnfs-bootclean.sh
S18kbd -> ../init.d/kbd
S19console-setup -> ../init.d/console-setup
S20alsa-utils -> ../init.d/alsa-utils
S20bootmisc.sh -> ../init.d/bootmisc.sh
S20x11-common -> ../init.d/x11-common
S21stop-bootlogd-single -> ../init.d/stop-bootlogd-single
```

***Figura 5.4, Elenco ordinato dei symlink e relativi scipt contenuti in /etc/rcS.d.***

Systemd invece, come visto in precedenza, presenta all'interno dei file target delle righe utili con l'etichetta "**Requires**" e "**Wants**"; qui sono definite le dipendenze dell'unità corrente rispetto alle altre.
Il sistema in questo modo conosce quali gruppi di attività avviare prima, completa la loro esecuzione e passa alle successive.

## 5.5 Funzionalità per il ripristino del sistema

Le più importanti funzionalità per l'analisi delle fasi di boot e per il controllo dei servizi interessati da tale processo sono già state trattate; una delle feature cardine di systemd che si differenzia dalle altre perché non risulta strettamente legata alla fase di init è lo **snapshotting** ed il **ripristino** dello stato del sistema.
È possibile salvare lo stato attuale del sistema con il comando **systemctl snapshot** che raccoglie le dipendenze delle unità attive e le memorizza temporaneamente in un file che deve terminare con *.snapshot*.
Il file è disponibile solo per la sessione corrente e verrà automaticamente eliminato in fase di reboot.

L'utilità di questa funzione emerge in fase di test/debug di nuovi servizi; dopo aver avviato o arrestato i servizi in questione è possibile tornare ad uno stato definito e stabile del sistema con il comando **systemctl isolate *MY_SNAPSHOT.snapshot*.**

## 5.6 Journald, tutti i log in un posto solo

La versione 38 di systemd ha introdotto un proprio sistema di log chiamato journal, controllato dal servizio **systemd-journald**; il servizio precedente syslog non è più necessario.

La differenza sostanziale è che i log ora sono in formato binario.

Il nuovo daemon raccoglie a sé tutti i messaggi di log delle diverse componenti di un sistema Linux; in passato, diversi sottosistemi intervenivano nella registrazione dei messaggi di log (le informazioni erano memorizzate in file text diversi).

Il risultato era che i registri avevano differenti livelli di dettaglio e un amministratore doveva visionare molti file, ognuno con struttura propria, per correlare le informazioni presenti e scovare il problema.

Journald memorizza tutti i messaggi in un unico posto, indipendentemente dal fatto che siano messaggi del SO o del livello applicativo.

Il file non può essere aperto con un editor di testo ma attraverso i comandi di gestione del daemon; esso può anche crescere in maniera indefinita ma è lo stesso servizio ad impostare una soglia massima e a cancellare i log vecchi per mantenere la percentuale di memoria (centrale o secondaria, si veda in seguito) utilizzata sotto una determinata soglia.

La directory usata è */run/log/journal* per la modalità non persistente, opzione con la quale le informazioni verranno perse al riavvio successivo, oppure */var/log/journal* se lo storage è persistente.

Tutti i parametri della configurazione principale sono in */etc/systemd/journald.conf* .

```
#  This file is part of systemd.##  systemd is free software; you can redistribute
#  it and/or modify it under the terms of the GNU Lesser General Public License as
#  published by the Free Software Foundation; either version 2.1 of the License, or
#  (at your option) any later version.
#
# Entries in this file show the compile time defaults.
# You can change settings by editing this file.
# Defaults can be restored by simply deleting this file.
#
# See journald.conf(5) for details.

[Journal]

#Storage=auto
#Compress=yes
#ForwardToSyslog=yes
 ...
```

***Figura 5.5, Contenuto del file di configurazione /etc/systemd/journald.conf***

La ***Figura 5.5*** mostra un estratto della configurazione standard, riconosciamo in esso alcuni parametri:

- Storage, seguito da quattro possibilità:
    1. “none”, il journaling viene spento, ogni messaggio di log ricevuto è ignorato;
    2. “volatile”, i dati vengono salvati in memoria centrale e sono disponibili temporaneamente;
    3. “persistent”, i dati vengono salvati in maniera persistente, se la cartella */var/log/journal* non esiste viene creata.
    4. “auto”, journald tenta di salvare i dati in modalità persistente in */var/log/journal*, se la cartella precedente non esiste (ad esempio il disco potrebbe risultare non scrivibile) il salvataggio dei log avviene in */run/log/journal* (la modalità cambia in “volatile”, i dati andranno persi al riavvio);
- Compress, se è attivo i dati che superano il limite imposto vengono compressi prima di essere memorizzati su disco;
- ForwardToSyslog, il parametro specifica se i messaggi ricevuti da systemd-journald devono essere inoltrati anche a syslogd, il parametro permette di evitare conflitti se sono attivi entrambi i servizi; il valore di default è “yes”.

# 6 L’adozione di systemd ed il relativo dibattito

La prima grande distribuzione GNU/Linux ad utilizzare di default systemd è Fedora, in particolare la versione 15 *Lovelock*, pubblicata il 24 maggio 2011.

Tra l’ottobre 2013 e il febbraio 2014 si è sviluppato un lungo dibattito, prima tra gli sviluppatori Debian all’interno della mailing list, in seguito tra tutti gli appassionati della comunità open-source.

Il tema comune era: **sostituire** o **meno** il vecchio sistema di init in favore di **systemd**.

## 6.1 Dibattito nella community Debian

La distribuzione Debian nel processo di adozione di systemd è considerata un punto cardine; dopo di lei molte altre distribuzioni decisero di sostituire il proprio processo di init con quest’ultimo.

La community Debian era molto attiva e le discussioni legate allo sviluppo del progetto erano all’ordine del giorno; alcune passavano maggiormente in sordina ed erano seguite solo dagli interessati mentre altre arrivavano al grande “pubblico”.

Secondo la lunga tradizione Debian, la decisione finale per problemi riguardanti un determinato package, veniva rimandata al manutentore del package.

Il caso “systemd” fece eccezione.

Nel periodo citato, i tempi per gli sviluppatori Debian erano stringenti, bisognava prendere una decisione definitiva sul processo di init in vista del rilascio della versione *Jessie*.

Bisognava considerare anche un altro fattore, in questo caso le caratteristiche stesse del nuovo init andavano ad influenzare la progettazione di altri package ed il design del sistema nel suo complesso.

Per queste ragioni il dibattito fu seguito con molto interesse dagli appassionati.

La comunità risultava divisa su quale fosse la strada da intraprendere e anche la prima proposta di risolvere la diatriba attraverso una votazione fra tutti i Debian developer non fu accolta (Corbet 2013).

L’ex capo progetto di Debian Stefano Zacchiroli descriveva la sua opposizione in questo modo: “*GRs should be used for societal and policy decisions. Using GRs for *technical* decision is stupid. We really need to stop thinking that every single member of the Debian project, just because he/she is a DD, has a clue on every single technical matter that go on in the project.”*[8]

[8] GR sta per general resolution: votazione fra tutti i Debian developers (DD).

Molte persone avevano assunto la stessa posizione di Zacchiroli, trovare una soluzione per il problema si stava dimostrando difficile.

Il secondo passaggio del dibattito, al fine di scegliere il software più opportuno, interessa il comitato tecnico: un'istituzione Debian per risolvere controversie di carattere tecnico, le cui decisioni erano molto rispettate tra gli sviluppatori.

I membri del comitato erano *Bdale Garbee, Russ Allbery, Keith Packard, Don Armstrong, Andreas Barth, Ian Jackson, Steve Langasek,* e *Colin Watson.*

Le critiche non mancarono ad arrivare nemmeno per questo ulteriore passaggio, infatti qualcuno fece notare come Steve Langasek e Colin Watson erano, in quel periodo, impiegati alla Canonical[9].

La suddetta società aveva pesantemente investito su un altro sistema di init **upstart** e secondo alcuni avrebbe potuto far pressione sui due membri del comitato.

Le critiche di lì a poco si spensero ed il lavoro della commissione poté continuare.

La decisione prevedeva una scelta tra cinque possibili opzioni:

- Mantenere l'esistente system V init;
- Adottare Systemd;
- Adottare Upstart;
- Adottare OpenRC;
- Supportare sistemi di init multipli con una configurazione o altro.

Gli sviluppatori Debian non scartavano nemmeno l'ulteriore ipotesi di ignorare systemd, visto prevalentemente come un progetto della *Red Hat*, ignorare upstart, fortemente sponsorizzato dalla Canonical e seguire una strada propria verso l'implementazione di un gestore per l'inizializzazione proprietario.

La lista con le argomentazioni tecniche a favore di un sistema o di un altro fu stilata ed analizzata dagli sviluppatori; da essa emergevano alcune problematiche interessanti.

Debian sviluppava distribuzioni basate su diversi kernel, **Debian GNU/kFreeBSD** era basata su FreeBSD kernel mentre **Debian GNU/Hurd** era basata su Hurd kernel; ciononostante né systemd né upstart supportavano kernel *non-Linux.*

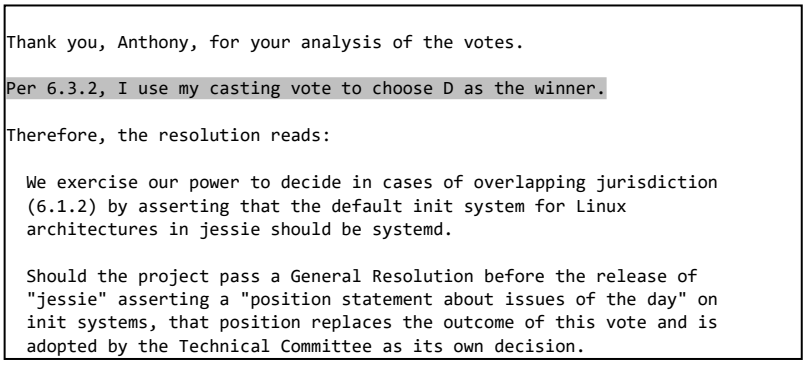
Quanta influenza avrebbero dovuto avere le distribuzioni *non-Linux* nella scelta finale era parte del dibattito.

L'elenco finale di argomentazioni, raccolto per ogni sistema di init proposto, è raggiungibile alla pagina https://wiki.debian.org/Debate/initsystem/.

[9] Società privata che sviluppa progetti legati al software libero, supporta lo sviluppo del sistema operativo GNU/Linux *Ubuntu*.

Dopo circa tre mesi di dibattito e due votazioni della commissione finite in stallo, il risultato finale fu di 4 preferenze per systemd e 4 preferenze per upstart. La contesa fu risolta solo con il voto di Bdale Garbee che ai tempi era il project Leader Debian; il suo voto in favore di systemd (***Figura 6.1***) e già conteggiato nel risultato finale, aveva valore doppio in caso di parità.

Il dibattito era concluso, il vincitore era systemd.



```
Thank you, Anthony, for your analysis of the votes.

Per 6.3.2, I use my casting vote to choose D as the winner.

Therefore, the resolution reads:

  We exercise our power to decide in cases of overlapping jurisdiction
  (6.1.2) by asserting that the default init system for Linux
  architectures in jessie should be systemd.

  Should the project pass a General Resolution before the release of
  "jessie" asserting a "position statement about issues of the day" on
  init systems, that position replaces the outcome of this vote and is
  adopted by the Technical Committee as its own decision.
```

***Figura 6.1, Messaggio nella mailing list Debian, Bdale Garbee dichiara vincitore systemd (D).***

## 6.2 Le scelte di Canonical

La distribuzione GNU/Linux attualmente più diffusa è *Ubuntu* sviluppata da Canonical.

Nel periodo tra il 2013 e il 2014 Ubuntu utilizzava la propria suite di init **upstart**, più veloce nell'eseguire il boot rispetto al classico init e anche altre distribuzioni facevano lo stesso.

Quando Debian decise di adottare systemd, gli sviluppatori di Canonical continuarono la strada intrapresa con upstart.

Per questo motivo, nel tempo, si creò una certa distanza fra lei e altre comunità di sviluppo.

Ubuntu in quel periodo stava contemporaneamente perfezionando il suo processo di avvio upstart, il display server **Mir** e l'ambiente desktop **Unity**.

Le principali distribuzioni rimanenti si stavano concentrando su systemd, sul display server Wayland e puntavano ad utilizzare GNOME o KDE come ambiente desktop.

L'isolamento di Ubuntu si mostrò subito dannoso, infatti allontanarsi dal resto della community significava mantenere la compatibilità dei propri moduli software (insieme alle patch e agli aggiornamenti necessari) solo con le proprie forze (Duckett 2014).

Lo sforzo era sicuramente troppo ingente per gli sviluppatori di Ubuntu che alla fine decisero di rendere compatibile il proprio OS con systemd e nell'Aprile 2015 di utilizzarlo di default sulla distribuzione ufficiale.

Ubuntu è l'ultimo, dei sistemi operativi più diffusi, che ha adottato systemd; per maggiori informazioni sì rimanda alla ***Figura 6.2***.

Il vecchio sysV init era ormai abbandonato, rimanevano aperti progetti alternativi a systemd che però non riscontravano una grande diffusione e non introducevano migliorie particolari.

| Linux distribution | Date added to software repository | Enabled by default? | Date released as default |
|---|---|---|---|
| Android | N/A (not in repository) | No | N/A |
| Arch Linux | January 2012 | Yes | October 2012 |
| CentOS | April 2014 | Yes | April 2014 (7.14.04) |
| Debian | April 2012 | Yes | April 2015 (v8) |
| Devuan | N/A (not in repository) | No | N/A |
| Fedora | November 2010 (v14) | Yes | May 2011 (v15) |
| Gentoo Linux | July 2011 | No | N/A |
| Mint | June 2016 (v18.0) | Yes | N/A |
| openSUSE | March 2011 (v11.4) | Yes | September 2012 (v12.2) |
| Red Hat Enterprise Linux | June 2014 (v7.0) | Yes | June 2014 (v7.0) |
| SUSE Linux Enterprise Server | October 2014 (v12) | Yes | October 2014 (v12) |
| Ubuntu | April 2013 (v13.04) | Yes | April 2015 (v15.04) |

***Figura 6.2, Tabella con i dati sull'adozione di systemd nelle principali distribuzioni***

# 7 Svantaggi e critiche legate a systemd

Le critiche al moderno gestore di sistema e dei servizi sono molteplici, alcune di carattere tecnico e riguardano una scelta progettuale ben specifica, altre invece sono opposizioni che interessano il sistema nel suo complesso e non fanno riferimento ad un determinato dettaglio. L'autore del pensiero, quando disponibile, sarà sempre presente; per completezza sono state riportate anche critiche non riconducibili ad una specifica persona, questo perché i siti di settore visionati mancavano di specificare alcuni riferimenti bibliografici.

## 7.1 Troppe responsabilità per un solo software

La prima critica che viene mossa nei confronti del progetto systemd è di non aver rispettato la filosofia Unix: *"do one thing and do it well"* nota anche con l'acronimo DOTADIW.
Il termine filosofia Unix sta ad indicare l'insieme di norme ed approcci che vengono seguiti per lo sviluppo software di sistemi operativi Unix-like e non solo. Le regole suddette hanno lo scopo di rendere il codice più semplice, chiaro, modulare e mantenibile; quando si lavora con progetti complessi, in cui operano interi team di sviluppo, questi semplici concetti assumono grande rilevanza.
Patrick Vokerding, fondatore della distribuzione Linux Slackware, insieme ad altri esponenti del mondo informatico pensa che systemd non abbia rispettato la norma DOTADIW perché le funzionalità del software sono troppe ed eccedono quelle di un normale sistema di init (Volkerding 2012).
Oltre alla fase di boot, systemd presenta feature per la gestione del risparmio energetico, la gestione dei dispositivi, la crittografia del disco, i log di sistema, la configurazione di rete, il login degli utenti, la gestione dei servizi e dei timer.
Un design di questo tipo è solitamente apprezzato dagli utenti che con un solo servizio possono controllare più caratteristiche del sistema, ma è "odiato" da molti sviluppatori che vedono in questa architettura una possibile *single point of failure*; ad esempio un bug presente in systemd può influire su tutti quei servizi che vengono avviati e gestiti da esso.
È necessario analizzare due aspetti legati a questa criticità.
La prima è legata agli aggiornamenti del sistema, immaginiamo ad esempio che debba essere aggiornato il codice di systemd relativo alla gestione dei dispositivi bluetooth, in una macchina che non presenta dispositivi di questo tipo attivi; l'aggiornamento costringerà l'utente ad un riavvio del sistema.

In generale, qualsiasi piccola modifica nel codice del software di init necessiterà di un riavvio del sistema; in passato, gli sviluppatori di distribuzioni Linux avevano mosso questa stessa critica al sistema operativo Microsoft Windows che per semplici upgrade non-kernel richiedeva un reboot completo.
Non dimentichiamo che il kernel Linux è utilizzato prevalentemente per sistemi embedded, server, supercomputer, tablet e smartphone; per quanto riguarda desktop e laptop possiede solo una piccola fetta di mercato (circa il 2-3% (Operating System Market Share 2019)). Una diffusione così eterogenea fa sì che alcune problematiche, come il già citato riavvio del sistema dopo aggiornamenti non critici, rappresentino una forte limitazione per quei calcolatori (ad esempio i server) che devono funzionare con continuità.
La seconda implicazione, sempre legata alle eccessive responsabilità di cui si fa carico systemd, è che molti servizi di terze parti hanno sviluppato dipendenza con systemd e le sue funzionalità.
Consideriamo l'ambiente desktop GNOME (GNU Network Object Model Environment) per i sistemi operativi GNU/Linux, esso deve interagire con alcune feature di systemd (principalmente servizi per il login e per la gestione dei log) per il corretto funzionamento.
La forte dipendenza tra software applicativi e il boot manager può portare a episodi spiacevoli; ad esempio, nel caso fosse introdotto un aggiornamento di systemd che provochi malfunzionamenti ad applicazioni terze, il team di sviluppo potrebbe non interessarsi al problema e non apportare nessuna correzione.

## 7.2 Implicazioni negative legate alla popolarità di systemd

La diffusione stessa di systemd è vista secondo alcuni esponenti della community del software libero come una criticità.
L'utilizzo da parte delle più importanti distribuzioni Unix-like ha trasformato systemd in uno standard de facto per quanto riguarda la gestione dei servizi e il boot del sistema. Riprendendo il già citato GNOME, esso ha deciso di sfruttare le API del nuovo processo di init proprio perché erano quelle più diffuse tra le diverse distribuzioni, costringendo altri software di avvio ad adottare le stesse API per questioni di compatibilità (Quora 2016).
Inoltre, alcune applicazioni potrebbero risultare sfavorite, rispetto ad altre, dai cambiamenti che systemd deciderà di introdurre nel tempo; questo è indirettamente un'altra violazione della Unix philosopy: "*Write programs to work together".*
Le dipendenze generate con il passare del tempo hanno creato una forte limitazione per gli utenti più esperti di molte distribuzioni GNU/Linux, dato che queste ultime permettono di

sostituire servizi tipici del sistema: a titolo di esempio citiamo elementi come il desktop environment (KDE, GNOME, Cinammon o altri) o il gestore grafico (X Window System, Wayland o ulteriori gestori), mentre invece risulta particolarmente complesso sostituire il sistema di init installato di default dalla distribuzione.

## 7.3 Inutile appesantimento del codice sorgente

Nel 2013 systemd includeva 69 file binari differenti racchiusi in un singolo repository e rilasciati seguendo un ciclo di rilascio unificato.

Uno dei servizi più discussi fu quello dei timer di systemd, che secondo alcuni, era un inutile appesantimento del codice dato che andava a svolgere la stessa funzione dell'utility *cron* già presente in tutti i sistemi.

Il software *cron* è uno scheduler basato sul tempo presente nei sistemi operativi Unix-like che permette di avviare comandi e shell script periodicamente in determinati orari, giorni o intervalli; il tutto per velocizzare e facilitare parte del lavoro dell'amministratore di sistema.

Ovviamente la sua introduzione portava anche vantaggi, in primis la gestione dei timer attraverso il nuovo gestore avveniva utilizzando i file unit (l'estensione .timer identifica proprio questo tipo di file) come per tutti gli altri servizi svolti da systemd.

I dettagli sugli eventi gestiti dai timer risultavano nei log di systemd-journal, i timer erano attivati/disattivati con un solo comando e le azioni da intraprendere venivano eseguite in tempi flessibili ma comunque ben definiti (ad esempio in base a cambiamenti di stato di dispositivi hardware oppure con riferimenti temporali del tipo: "5 minuti dopo il boot del computer").

## 7.4 Stretta dipendenza con il kernel Linux

Nel presente trattato è già emerso come il processo di init moderno si appoggi su funzioni esclusive del kernel Linux come i cgroup e le socket, abbiamo anche visto come nel dibattito Debian parte della community disapprovava systemd a causa del suo stretto legame con Linux; infatti i Debian developer continuavano a portare avanti anche distribuzioni con kernel *non-Linux*.

Il dibattito sulla forte dipendenza systemd-Linux si è attenuato con il tempo, il perché è semplice, il kernel Linux è ormai considerato stabile e per molti aspetti insostituibile; oggi giorno è installato su più del 60% dei server e sulla maggior parte dei dispositivi mobile (smartphone e tablet Android utilizzano Linux).

Lo stesso ideatore di Linux, Linus Torvalds, in una recente intervista, di cui viene riportato un passaggio in ***Figura 7.1,*** ha espresso il suo pensiero sul moderno gestore di sistema dichiarando di non notare grandi problemi in systemd seppur non condivida alcuni dettagli implementativi come l'uso di log binari.

*"When it comes to systemd, you may expect me to have lots of colourful opinions, and I just don't. I don't personally mind systemd, and in fact my main desktop and laptop both run it.*
*Now, I don't get along with some of the developers and think they are a bit too cavalier about bugs and compatibility, but I'm also very much not in the camp of people who hate the very thought of systemd."*
*...*
*"Now, I'm still old-fashioned enough that I like my log-files in text, not binary, so I think sometimes systemd hasn't necessarily had the best of taste, but hey, details…"*

***Figura 7.1, Parte dell'intervista di Linus Torvalds su systemd, intervista IT wire***

## 7.5 Critiche ulteriori

Una critica diffusa in passato e poi smentito dallo stesso Lennart Poettering è che l'ampia diffusione della suite software sia legata all'influenza di Red Hat[10]; 6 sviluppatori incluso lo stesso Poettering erano impiegati alla Red Hat nel primo periodo di sviluppo mentre altri provenivano da Debian, Intel, ArchLinux, Canonical, Mandriva e Pantheon.

Un ultimo aspetto critico riguardava la manutenibilità del progetto, a causa della grandezza del codice e delle relazioni che interconnettono profondamente le varie parti; era opinione diffusa che, estendere e gestire il codice di systemd, potesse risultare difficile se non si faceva parte di quel gruppo di programmatori che ha lavorato allo sviluppo sin dalla prima versione. Poettering smentì anche questa voce sottolineando la modularità del codice (in fase di compilazione è possibile scegliere quali pacchetti software compilare), la documentazione dettagliata che accompagna lo sviluppo ed il tentativo marcato di semplificare il debugging (venivano forniti componenti per l'esecuzione verbosa, per il debugging interattivo insieme ad altri strumenti di diagnostica).

L'ideatore di systemd scrisse anche un post, nel gennaio 2013, dal nome **The Biggest Myths** nel quale sfatò molte delle dicerie che in quel periodo affollavano forum e mailing list (Poettering, The Biggest Myths 2013).

[10] Società multinazionale statunitense che si dedica allo sviluppo software, acquisita da IBM nel luglio del 2019.

# Conclusioni

Il presente elaborato ha analizzato il procedimento di inizializzazione di sistemi operativi Unix-like con Kernel Linux; in particolare sono stati trattati i software di init: System Unix V init e systemd.

Abbiamo visto come il primo è stato sostituito quasi completamente dal secondo, dopo che il dibattito sviluppato all'interno delle community di sviluppatori ha decretato systemd come miglior suite software per l'inizializzazione del sistema.

L'argomento è stato esaminato da diversi punti di vista, alternando descrizioni generali dei processi a spezzoni di codice/file di configurazione che fanno riferimento all'implementazione stessa.

La trattazione proposta non vuole essere completa: per ogni dettaglio sull'utilizzo dei software citati, si rimanda ai manuali disponibili online.

Per quanto riguarda l'implementazione, essa è stata trattata in maniera limitata per ovvie ragioni; ma è importante notare come tutti i soggetti visti in azione, Kernel Linux, System V init e systemd siano progetti open source.

Infatti, gran parte del codice citato è sotto la licenza GNU General Public License versione 2 che garantisce le quattro libertà fondamentali del software: eseguire, studiare, ridistribuire e migliorare il codice.

Forte di questo aspetto, un elaborato futuro potrebbe analizzare l'implementazione che questi software nascondono e proporre migliorie per ottimizzare la delicata, ma fondamentale, fase di init.

Sempre per quanto riguarda l'implementazione, potrebbe risultare interessante un'analisi comparativa tra systemd e launchd, software di init usato dalla Apple per macOS, in grado di mostrare le differenze di design ed i principi che le hanno motivate.

# Bibliografia e Sitografia

*Boot.* 3 luglio 2019. https://it.wikipedia.org/wiki/Boot (consultato il giorno 8 27, 2019).

Both, David. *linux-boot-and-startup.* 20 febbraio 2017. https://opensource.com/article/17/2/linux-boot-and-startup (consultato il giorno 8 28, 2019).

Chaiken, A. *analyzing-linux-boot-process.* 16 gennaio 2018. https://opensource.com/article/18/1/analyzing-linux-boot-process (consultato il giorno 8 28, 2019).

Corbet, J. *Which init system for Debian.* 5 novembre 2013. https://lwn.net/Articles/572805/.

Duckett, C. *Debian init decision further isolates Ubuntu.* 12 febbraio 2014. https://www.zdnet.com/article/debian-init-decision-further-isolates-ubuntu/.

Jones, M. *Inside the Linux boot process - IBM developer.* 31 maggio 2006. https://developer.ibm.com/articles/l-linuxboot/.

Negus, C. *Linux Bible 8th Edition.* Indianapolis: John Wiley & Sons, 2012.

*Operating System Market Share.* agosto 2019. https://netmarketshare.com/operating-system-market-share.

Poettering, L. *Rethinking PID 1.* 30 aprile 2010. http://0pointer.de/blog/projects/systemd.html (consultato il giorno 9 3, 2019).

—. *systemd for administrator.* 27 giugno 2012. http://0pointer.net/blog/projects/self-documented-boot.html.

—. *The Biggest Myths.* 26 gennaio 2013. http://0pointer.de/blog/projects/the-biggest-myths.html.

*Quora.* 26 giugno 2016. https://www.quora.com/Why-does-GNOME-3-require-systemd.

*Systemd.* 2 agosto 2019. https://en.wikipedia.org/wiki/Systemd (consultato il giorno 8 29, 2019).

«systemd manpages.» *www.freedesktop.org.* s.d. https://www.freedesktop.org/software/systemd/man/index.html (consultato il giorno luglio 21, 2019).

*UEFI - wikipedia.* 7 settembre 2019. https://it.wikipedia.org/wiki/Unified_Extensible_Firmware_Interface.

Volkerding, P. *linuxquestions - Interview with Patrick Volkerding of Slackware.* 6 luglio 2012. https://www.linuxquestions.org/questions/interviews-28/interview-with-patrick-volkerding-of-slackware-949029/.

# Indice delle Figure