Università degli studi di Padova

Dipartimento di Tecnica e Gestione dei Sistemi Industriali

Corso di Laurea Triennale in
Ingegneria Meccanica e Meccatronica

LA DIFFUSIONE DEL SOFTWARE OPEN-SOURCE NEI SISTEMI EMBEDDED

THE SPREAD OF OPEN-SOURCE SOFTWARE IN EMBEDDED SYSTEMS

RELATORE: CH.MO PROF. Reggiani Monica

LAUREANDO: Vellere Matteo

ANNO ACCADEMICO: 2013/2014

INDICE GENERALE

INTRODUZIONE

In questo momento storico i Sistemi Embedded hanno assunto un ruolo pervasivo in tutti gli ambiti della società. Sempre di più vediamo l'integrazione e il perfezionamento di apparecchi che utilizziamo tutti i giorni, a cui vengono aggiunte funzionalità che poco tempo fa pensavamo impensabili. Un semplice frigorifero, che fino a ieri era un contenitore di alimenti in grado di mantenere una temperatura più bassa di quella esterna, oggi si è dotato di schermi a led e di funzioni elettroniche che ne controllano il corretto funzionamento. Non si può escludere che domani lo stesso frigorifero possa autonomamente ordinare la spesa via internet in base a quello che manca. Un altro esempio sono i telefoni cellulari che di mese in mese si evolvono e impiegano le più moderne tecnologie in campo elettronico, passando da “cellulari” a “smartphones” in pochi anni. Siamo testimoni di una vera e propria rivoluzione digitale e ancora non sappiamo dove ci porterà questo progresso.
Nei sistemi embedded, di cui verrà fornita una definizione precisa nel primo capitolo, si ha l'integrazione di elettronica ed informatica creando una sinergia tra hardware e software per svolgere le funzioni per cui sono progettati. In questo campo mosso da una fortissima spinta tecnologica, si stanno affermando sempre più soluzioni software open source che dimostrano di avere una marcia in più rispetto agli strumenti tradizionali, non solo per il basso costo iniziale ma soprattutto per l'intero ecosistema che ruota loro attorno. Le soluzioni open source si dimostrano orientate verso il futuro, con un occhio di riguardo rivolto all'efficienza, al risparmio energetico e alle prestazioni.

L'intento principale con cui ho scritto questa tesi è quello di fornire una panoramica generale su quali sono, al giorno d'oggi, i principali strumenti Open Source che vengono impiegati nei sistemi embedded. Ho cercato, con l'aiuto di studi di settore e sondaggi, di presentare un panorama che di per sé è molto complesso e faticosamente delineabile, data la sua essenza mutevole e dato lo stato attuale di un settore in forte evoluzione come quello dei sistemi embedded.
La tesi si articolerà nei seguenti punti:

- Nel primo capitolo viene data una definizione di sistema embedded, viene fornito un breve excursus storico del settore e vengono presentate le caratteristiche generali delle piattaforme su cui i sistemi embedded sono implementati.
- Nel secondo capitolo si presenta il software Open Source dalle sue origini, considerando la filosofia alla base del movimento e le implicazioni etiche e sociali, il

vantaggi che ne derivano, la sua diffusione e le principali licenze utilizzate in questo campo.

- Nel terzo capitolo si è cercato di unire i contenuti dei primi due capitoli trattati fornendo la casistica della realtà odierna sul software Open Source nei sistemi embedded, dando ampio spazio ai grafici aggiornati sull'impiego dei sistemi operativi e analizzando i vantaggi di questa scelta.
- Il quarto e il quinto capitolo sono dedicati rispettivamente a Linux Embedded e ad Android per fornire i dettagli, la struttura e le caratteristiche di due dei sistemi operativi più largamente utilizzati negli embedded.

CAPITOLO 1

1.I Sistemi Embedded

1.1.Generalità sui Sistemi Embedded

"In elettronica ed informatica, con il termine **sistema embedded** si identificano genericamente tutti quei sistemi elettronici di elaborazione a microprocessore progettati appositamente per una determinata applicazione (*special purpose*). Possono essere sistemi non riprogrammabili dall'utente per altri scopi, spesso con una piattaforma hardware *ad hoc*, integrati nel sistema che controllano ed in grado di gestirne tutte o parte delle funzionalità richieste" [1].

Già da questa breve definizione possiamo intuire quanto diffuso sia l'impiego di sistemi di questo tipo e quanti dispositivi di uso quotidiano rientrano nella categoria. Una lista, non esaustiva, di quello che può essere considerato un sistema embedded è presentata nel seguito:

- Personal computer dedicati all'automazione industriale e il controllo di processo.
- Sportelli Bancomat e apparecchi POS.
- Elettronica aeronautica, come sistemi di guida inerziale, hardware/software di controllo per il volo e altri sistemi integrati nei velivoli e nei missili.
- Telefoni cellulari.
- Centralini telefonici.
- Apparecchiature per reti informatiche come router, timeserver e firewall, switch.
- Stampanti e Fotocopiatrici.
- Sistemi di stoccaggio di dati come hard disk, floppy disk o compact disc.
- Sistemi di automazione casalinghi come termostati, condizionatori e altri sistemi di monitoraggio della sicurezza.
- Distributori di bevande.
- Elettrodomestici come forni a microonde, lavatrici, apparecchi televisivi, lettori o scrittori di DVD, ma anche un banale spazzolino elettrico di alta fascia.
- Apparecchiature biomedicali come ecografi, scanner medici per risonanza magnetica.
- Equipaggiamenti medici.
- Strumenti di misura come oscilloscopi digitali, analizzatore logico, e analizzatore di spettro.

- I PLC (Programmable Logic Controller) utilizzati per l'automazione industriale.
- Console per videogiochi fisse e portatili.
- Centraline di controllo dei motori automobilistici e degli ABS.
- Strumenti musicali digitali quali tastiere workstation, mixer digitali o processori audio.
- Centraline di controllo nelle automobili, autocarri e motocicli (motore, illuminazione, assistenza alla guida, ecc).
- Decoder per TV digitale.
- Sistemi per la domotica.

Illustrazione 1: Esempi di sistemi embedded

1.2.Evoluzione storica dei dispositivi

Il primo sistema embedded moderno viene considerato l'Apollo Guidance Computer (AGC), sviluppato da Charles Stark Draper presso il MIT Instrumentation Laboratory nel 1961. Per ogni volo lunare veniva utilizzato per controllare il Command Module e il Lunar Module. L'AGC è uno dei primi sistemi che possono essere considerati IC (integrated circuit) e, al momento della concezione, l'Apollo Guidance Computer era considerato uno dei più rischiosi oggetti all'interno del progetto Apollo nonostante i suoi 32 kg di peso e le sue dimensioni piuttosto contenute (61x32x17cm). Era basato su una unità di calcolo da 2 MHz di velocità di clock, di 2 Kwords di memoria RAM e di una trentina abbondante di Kwords di memoria ROM. Quest'ultima conteneva, principalmente, dati e programmi. L'AGC era multitasking, essendo in grado di eseguire fino a 8 programmi contemporaneamente. Nella sua prima

versione utilizzava 4.100 chip di porte logiche NOR. Nella seconda versione, che utilizzava due porte logiche NOR all'interno di ogni integrato, il numero complessivo di chip fu ridotto a 2.800. Appunto l'utilizzo dei nuovi circuiti integrati monolitici per ridurne la taglia e il peso, aumentò considerevolmente il rischio.

Sempre nel 1961 fu il turno del sistema computerizzato di guida Autonetics D-17 prodotto per la prima volta su larga scala per il missile Minuteman. Utilizzava circuiti logici realizzati con transistor e un hard disk come memoria principale. Quando il Minuteman II venne prodotto nel 1966, il sistema di guida D-17 fu rimpiazzato da un nuovo computer che utilizzava dei circuiti integrati, e che fu il primo caso di utilizzo di tali componenti in grandi volumi, contribuendo ad abbassarne i prezzi e a diffonderne, di conseguenza, l'utilizzo. In seguito, sempre grazie alle ricerche in campo militare, i sistemi embedded hanno subito una riduzione dei costi così come una enorme crescita della capacità di calcolo e delle loro funzionalità. La cosiddetta Legge di Moore si è rivelata esatta in tutto questo periodo. Nel 1972 arrivò il primo microprocessore commercializzato, l'Intel 4004, aveva una frequenza di clock di 740 kHz e venne montato su calcolatrici ed altri sistemi di piccole dimensioni. I prezzi erano in caduta libera e sempre più applicazioni cominciarono ad adottare un approccio a microprocessore, piuttosto che metodologie di progetto di circuiti logici personalizzate. Verso la metà degli anni ottanta, un maggiore grado di integrazione permise il montaggio di altri componenti, in precedenza collegati esternamente, sullo stesso chip del processore. Questi sistemi integrati vennero chiamati microcontrollori piuttosto che microprocessori e fu possibile il loro utilizzo di massa. Ci fu un'esplosione del numero di sistemi embedded distribuiti sul mercato, così come delle componenti fornite da vari produttori per facilitare la progettazione di tali sistemi. Già verso la fine degli anni ottanta, i sistemi embedded rappresentavano la regola piuttosto che l'eccezione per quasi tutti i dispositivi elettronici. Successivamente i sistemi embedded andarono a svolgere quei compiti per cui non era più possibile impiegare i tradizionali

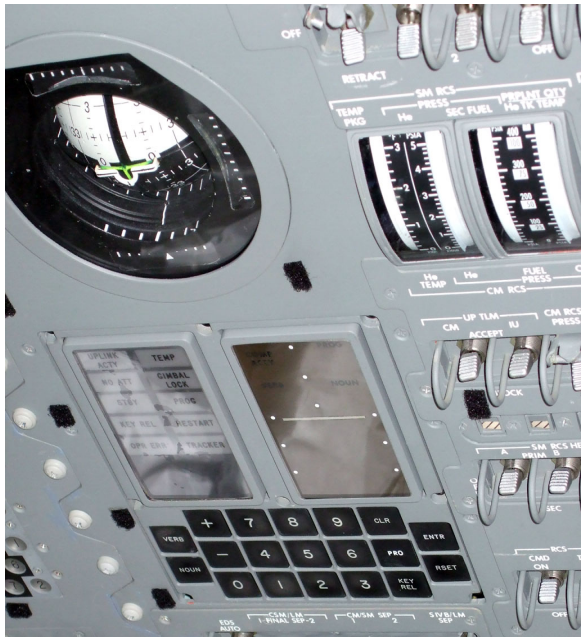


Illustrazione 2: L'interfaccia DSKY (display-Keyboard) dell'AGC sul pannello del modulo di comando Apollo.

computer general purpose, conquistando sempre più fette di mercato. Negli ultimissimi anni per i sistemi embedded vengono impiegati il 98% della totalità dei chips prodotti lasciando solamente il 2% al mondo dei PC [2]. Secondo le statistiche del World Trade Office i sistemi embedded, stimati in numero di 8 miliardi nel 2000, sono cresciuti fino a raggiungere i 16 miliardi nel 2010, con una proiezione di crescita che li vedrà toccare i 40 miliardi nel 2020 [3].

1.3.Caratteristiche

Come è stato illustrato i sistemi embedded si possono trovare ovunque negli oggetti quotidiani, negli elettrodomestici, nei complessi sistemi di automazione industriale, nei sistemi di controllo qualità, ecc. Facendo un paragone con i classici PC i sistemi embedded normalmente presentano una minore potenza di calcolo e una memoria limitata. Spesso però, per i compiti che devono svolgere, quelle caratteristiche che possono sembrare delle carenze diventano meriti: i sistemi embedded infatti sono più economici e più facili da progettare. La semplicità di progettazione si riflette in modo equo sia sull'hardware che sul software. La specializzazione che hanno per risolvere solo un preciso compito permette, ad esempio, un implementazione attraverso un OS semplificato di operazioni real-time (concetto approfondito nel paragrafo 4.2.2) oppure il funzionamento senza un sistema operativo.
Le piattaforme hardware dove vengono implementati i sistemi embedded includono microcontrollori, microprocessori, FPGA, Digital Signal Processor (DSP), e circuiti integrati per la specifica applicazione (ASIC), ciascuno con le proprie peculiarità. Le differenti piattaforme sono trattate nello specifico nel paper *Comparison of Embedded System Design for Industrial Applications* di Aleksander Malinowski e Hao Yu [4], del quale di seguito si fornirà un breve riepilogo.

1.3.1.Microcontrollori e Microprocessori

Per molti anni microcontrollori e microprocessori sono stati l'unico supporto hardware efficiente dove si potesse implementare un sistema embedded, grazie alle loro funzionalità programmabili. L'architettura di un microprocessore è fissa e generica il che mantiene il prezzo basso. Essi sono in grado di eseguire una sequenza di istruzioni tipicamente memorizzate in memorie a sola lettura (ROM) o in più recenti memorie Flash. I microcontrollori sono tipicamente costituiti da una memoria, alcune periferiche analogiche e digitali e un processore integrato. Per mantenere un costo basso e ridurre la potenza consumata, alcuni microcontrollori sono progettati per usare istruzioni della lunghezza di soli 4 bit. Essi possiedono una piccola RAM ed una frequenza di clock dell'ordine dei kHz. D'altro

canto esistono anche microcontollori che utilizzano istruzioni a 32 o 64 bit, clock dell'ordine delle centinaia di mega Hz e una potenza di calcolo paragonabile ad un DSP. In ogni caso ROM interne e piccole quantità di RAM sono integrate nel chip.

1.3.2.Field Programmable Gate Array (FPGA)

Le FPGA sono state sviluppate per i sistemi embedded basandosi sull'idea di creare degli array di blocchi logici contornati da blocchi di I/O, il tutto assemblabile secondo esigenze specifiche. Il vantaggio delle FPGA sta nella velocità di esecuzione, nella capacità di riconfigurazione, nell'ampio numero di componenti e nella varietà dei protocolli supportati. Nei sistemi embedded sono impiegate in due modi: per implementare funzionalità specifiche direttamente in logica digitale o per implementare l'architettura di un microprocessore. L'interesse per questi dispositivi si è accentuato recentemente perché i prezzi si sono abbassati arrivando a competere con i microcontrollori. L'uso delle FPGA permette anche un facile design di periferiche hardware personalizzate, come testimonia il paper di *Rodriguez-Andina et al.* sullo studio dell'evoluzione delle caratteristiche di questi dispositivi [5].

1.3.3.Digital Signal Processors (DSP)

I DSP sono progettati per avere al loro interno dei blocchi moltiplicatori e dei blocchi di analisi del segnale che consentono di eseguire complesse operazioni aritmetiche, il che gli rende ideali per l'implementazione tramite una programmazione di alto livello. Confrontandoli con i microcontrollori il vantaggio dei DSP è la possibilità di avere in un unico ciclo le operazioni di moltiplicazione e memorizzazione. I DSP hanno anche capacità di calcolo in parallelo e blocchi di memoria integrati, che ne aumentano la velocità. Alcune architetture simili ai DSP ma chiamate Digital Signal Controllers (DSC) sono ottimizzate per il controllo e contengono anche periferiche che possono, ad esempio, generare una PWM. In confronto alle FPGA i DSP hanno un costo maggiore e usualmente vengono impiegati dove si necessita di una grande potenza computazionale ad esempio quando serve processare immagini o video anche se l'applicazione industriale principale si ha nei controlli per motori.

1.3.4.Application Specific Integrated Circuit (ASIC)

Un'ultima tipologia di piattaforma su cui si possono implementare sistemi embedded sono gli ASIC, che forniscono performance di alta qualità, un piccolo consumo energetico e basso costo. Sono composti da *standard cells* e negli anni la complessità e le funzionalità che

assicurano sono state in continuo aumento. Lo stato attuale include blocchi con processori a 32bit, ROM, RAM, EEPROM, memorie Flash e altri blocchi complessi. L'impiego di questa tecnologia avviene in applicazioni di elevata qualità e dove si richiedono alte prestazioni per recuperare un costo di ingegnerizzazione iniziale che è più alto rispetto ad altre piattaforme.

CAPITOLO 2

2.Il Software Open Source

2.1.Disambiguazione

È necessario fare una distinzione per quanto riguarda la terminologia e separare i concetti di *open source* da *software libero* (*free software*) e *freeware,* per non commettere errori comuni. Va sottolineata l'ambivalenza che ha la parola “free” in lingua inglese, infatti si può intendere “free” nel senso di gratuito, gratis o che non richiede pagamento come ad esempio: “would you like a free beer?” (“gradiresti una birra gratis?”). Se dobbiamo trasportare questo concetto nel campo dei software possiamo quindi parlare di *freeware*. Un software di qualsiasi genere si dice *freeware* se viene rilasciato gratuitamente, spesso però con i sorgenti non disponibili. Per questo esso non ha nulla a che vedere con i concetti di *open source* e di *software libero.* All'interno della tesi il concetto di *freeware* non verrà approfondito ulteriormente. Il secondo modo in cui possiamo intendere la parola “free” è nel senso di libero (“Free as in Freedom” [6]) o di libertà nel senso più alto del termine. Una particolare attenzione si può prestare anche ai significati sociologici e filosofici che questa parola si porta dietro, come verrà approfondito meglio nel paragrafo successivo. *Free software,* o in italiano *software libero*, si riferisce a tutti quei software rilasciati con una specifica licenza che permette a ognuno di utilizzarli e ne incoraggia lo studio e la distribuzione. E' inoltre l'espressione che usa il fondatore della Free Software Foundation Richard M. Stallman quando si riferisce alle iniziative del suo movimento. Con *Open Source* software (i cui canoni rispondono alla *Open Source Definition* dell'Open Source Initiative) ci si riferisce sempre a quei programmi o prodotti informatici il cui codice sorgente è libero e, come nel *Free Software,* ne viene favorito lo studio e l'apporto di modifiche. Questo termine però è stato coniato successivamente per distanziarsi dal movimento di Stallman e, mentre i contenuti sono sostanzialmente gli stessi, quello che cambia è l'approccio: parlando di *software libero* si pone l'accento sugli aspetti sociologici ed etici che sono volutamente rimossi nella visione *open source* [7], la quale si concentra invece sui vantaggi pratici e tecnici della condivisione dei sorgenti.

2.2.Origine

Considerato che la condivisione del codice è nata insieme all'informatica piuttosto che di origini dell'Open Source, potrebbe essere più appropriato parlare invece di origine del

software proprietario ed esaminare il contesto storico in cui questa origine ha avuto luogo. Con la diffusione infatti dei primi PC presso le università lo scambio di software e di idee in ambito accademico era considerata una prassi. Solo con l'inizio degli anni ottanta quando lo sviluppo del software cominciò a passare di mano dalle università alle aziende si ebbe un inversione di tendenza. Inizialmente il business delle aziende era incentrato sull'hardware dato che questo costituiva la maggior parte degli introiti e la parte più costosa del prodotto, il software era considerata una parte accessoria. Le aziende stesse favorivano la diffusione del software, i cui sorgenti erano pubblici, per poter vendere un prodotto accompagnato da più software possibili e per essere più concorrenziali e soddisfare le esigenze sempre crescenti nel campo di un mercato in continua espansione. Con il passare del tempo il software diventò sempre più complesso e difficile da realizzare e le aziende iniziarono a non distribuire i sorgenti e obbligare i propri dipendenti a non rivelare nulla per non avvantaggiare la concorrenza; inoltre con il crollo dei costi dell'hardware, lo sviluppo commerciale del software divenne un business notevole, ed il codice sorgente un investimento prezioso che poteva da un lato far acquisire una fetta di tale mercato in rapida crescita e dall'altro legare gli utenti al proprio software mantenendo il segreto sui metodi utilizzati per lo sviluppo di sistemi e applicazioni. L'introduzione dei sistemi operativi rese i programmi sempre più portabili, in quanto lo stesso sistema operativo veniva offerto dal produttore di diversi modelli di hardware. La presenza di sistemi operativi funzionanti per macchine di differenti produttori hardware ampliava ulteriormente le possibilità di riutilizzo dello stesso codice e dunque l'utilità nell'impedire la duplicazione non autorizzata dei programmi da parte delle aziende. In questo contesto, e contemporaneamente con l'avanzare del lavoro dell'AT&T sul sistema operativo UNIX, molti programmatori (fra i quali Richard Stallman, che sarebbe diventato il portabandiera del software libero) si rifiutarono di lavorare per una società privata. Stallman fondò nel 1985 la Free Software Foundation (FSF), una organizzazione senza fini di lucro per lo sviluppo e la distribuzione di software libero. In particolare Stallman si dedicò allo sviluppo di un sistema operativo completo, compatibile con UNIX, ma distribuito con una licenza permissiva, con tutti gli strumenti necessari altrettanto liberi. Si tratta del progetto nato l'anno precedente, ovvero GNU, acronimo ricorsivo per contemporaneamente collegarsi e distinguersi da UNIX, ovvero "GNU is Not UNIX". *«L'obiettivo principale di GNU era essere software libero. Anche se GNU non avesse avuto alcun vantaggio tecnico su UNIX, avrebbe avuto sia un vantaggio sociale, permettendo agli utenti di cooperare, sia un vantaggio etico, rispettando la loro libertà.»*[6]. All'inizio degli anni novanta, il progetto GNU non aveva ancora raggiunto il suo obiettivo principale, mancando di completare il kernel del suo sistema

operativo. Nel 1991 arrivò quasi per caso il tassello mancante. Linus Torvalds, studente al secondo anno di informatica presso l'Università di Helsinki, decise di sviluppare per hobby un proprio sistema operativo imitando le funzionalità di Unix su un PC con un processore Intel 386. Torvalds era spinto dall'insoddisfazione riguardante alcuni applicativi di Minix (un sistema Unix-like su piattaforma PC), dal desiderio di approfondire le proprie conoscenze del processore Intel 386, e dall'entusiasmo per le caratteristiche tecniche di Unix. Torvalds distribuì il proprio lavoro tramite Internet e ricevette immediatamente un ampio riscontro positivo da parte di altri programmatori, i quali apportarono nuove funzionalità e contribuirono a correggere errori riscontrati. Nacque così il kernel Linux, il quale fu subito distribuito con una licenza libera e costituì il completamento perfetto per i tools di Stallman unendosi con esso in un sistema operativo chiamato GNU/Linux [8].

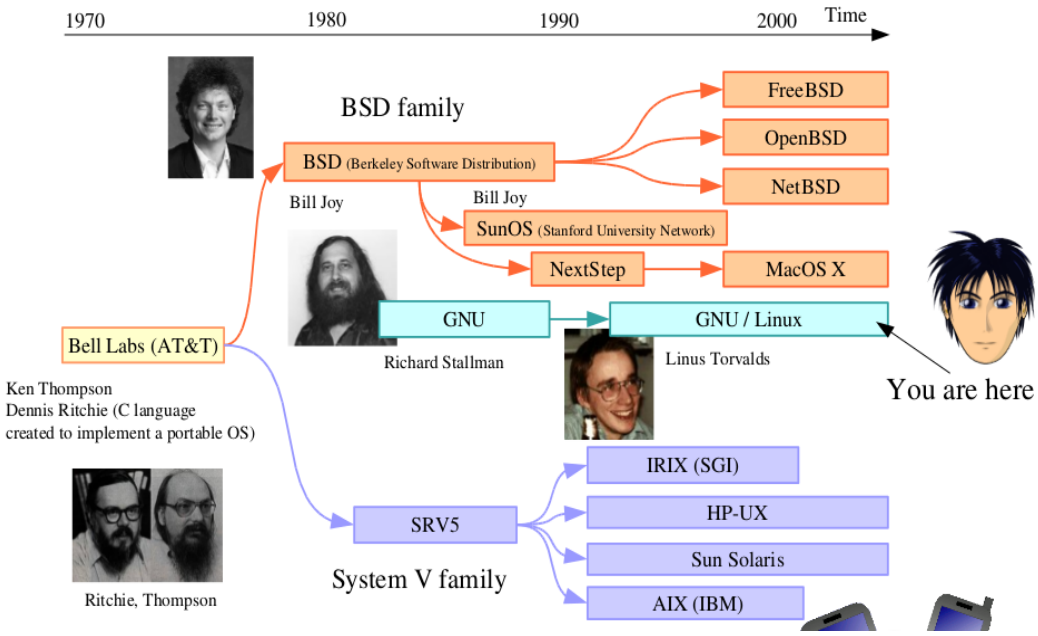


Illustrazione 3: Albero genealogico dei sistemi Unix

GNU/Linux oggi è alla base di moltissime distribuzioni create da comunità di sviluppatori o società. Linux è diventato sinonimo di open source e rappresenta il primo progetto di questo tipo ad aver avuto successo. Facendo affidamento essenzialmente sulla collaborazione via Internet di vari sviluppatori e appassionati si è dimostrato come un approccio decentralizzato possa funzionare senza un ordine apparente raggiungendo risultati ottimi velocemente. Eric S. Reymond nel suo saggio *La Cattedrale e il Bazaar* distingue l'approccio classico dello sviluppo software con quello comunitario di Torvalds paragonando rispettivamente l'uno con la costruzione pignola di una cattedrale da parte di pochi geni che lavorano in isolamento secondo uno schema preciso; e l'altro, ad un caotico bazaar dove tutti possono intervenire

ad apportare modifiche e migliorie, e dove il rilascio di nuove versioni è veloce e continuo [9]. Questo secondo approccio è tipico dello sviluppo del software open source e successivamente se ne analizzeranno tutti i pregi e le caratteristiche.

2.3.Filosofia

La filosofia del software libero si deve soprattutto ai numerosi trattati di carattere etico di Richard M. Stallman, fondatore della Free Software Foundation e portabandiera del movimento del software libero. Stallman si è battuto fin da studente per la libertà degli utenti nell'usare il proprio computer rifiutandosi di sottoscrivere accordi di non divulgazione con le società che vedevano nei laboratori universitari una fucina di talenti da circuire per accaparrarsi i profitti derivati dal loro lavoro. Il suo pensiero a tratti rivoluzionario ed utopistico si può riassumere nelle quattro Libertà Fondamentali che il software libero deve garantire :

Illustrazione 4: Lo stravagante e singolare Richard Matthew Stallman

1. Libertà di usare il programma senza impedimenti;
2. Libertà di aiutare se stesso studiando il codice disponibile e modificandolo in base alle proprie esigenze;
3. Libertà di aiutare il tuo vicino, cioè la possibilità di distribuire copie del software rielaborato, rendendolo così accessibile a tutti;
4. Libertà di pubblicare una versione modificata del software;

Solo rispettando queste quattro libertà e quindi conseguentemente solo sottoscrivendo opportune licenze si può essere certi di conservare la libertà dell'utente e, secondo Stallman, di poter contribuire alla salvaguardia del bene della società. Il software per Stallman è uno strumento per poter fare qualcos'altro per cui è basilare poterlo utilizzare completamente senza restrizioni, è basilare poterlo “prestare” ad un amico che ne ha bisogno per qualcosa di simile ed è ugualmente importante poterlo modificare per svolgere al meglio il proprio compito. Questa concezione utilitaristica del software come un mezzo per fare qualcosa di pratico e non come proprietà personale di chi lo scrive pone l'accento su un concetto diverso del diritto d'autore conosciuto attualmente nelle vesti del copyright. Stallman stesso conia il termine di *copyleft*, allo stesso tempo gioco di parole e concetto opposto al copyright, istituendo un

modello di gestione dei diritti d'autore basato su un sistema di licenze attraverso le quali l'autore (in quanto detentore originario dei diritti sull'opera) indica ai fruitori dell'opera che essa può essere utilizzata, diffusa e spesso anche modificata liberamente, pur nel rispetto di alcune condizioni essenziali. Nella versione pura e originaria del copyleft la condizione principale obbliga i fruitori dell'opera, nel caso vogliano distribuire l'opera modificata, a farlo sotto lo stesso regime giuridico. In questo modo, il regime di copyleft e tutto l'insieme di libertà da esso derivanti sono sempre garantiti. Una licenza basata sui principi del copyleft trasferisce a chiunque possegga una copia dell'opera alcuni dei diritti propri dell'autore. Inoltre consente la redistribuzione dell'opera stessa solo se tali diritti vengono trasferiti assieme ad essa. Fondamentalmente, questi diritti sono le quattro "libertà fondamentali" indicate da Stallman.
Il pensiero filosofico del movimento del software libero afferma anche che quando non sono gli utenti a controllare il programma, allora il programma controlla gli utenti; e gli sviluppatori controllano il programma, quindi attraverso di esso controllano gli utenti. Un tale programma, non-libero o “proprietario”, diventa quindi uno strumento di abuso.

« Ho avviato il movimento del software libero per rimpiazzare con software libero rispettoso della libertà, il software non libero che controlla l'utente. Con il software libero, possiamo almeno avere il controllo su quel che il software fa nei nostri computer. »
(Richard Stallman, The Anonymous WikiLeaks protests are a mass demo against control)

Stallman stesso è stato criticato per eccessivo estremismo nel suo pensiero radicale proprio per la visione delle compagnie di software non-libero come il male assoluto e le sue idee venivano viste con sospetto dall'ambiente commerciale statunitense, il che non facilitava la diffusione del software libero. Per favorire dunque l'idea delle licenze liberali nel mondo degli affari, Bruce Perens, Eric S. Raymond, Sam Ockman e altri cominciarono nel 1997 a pensare di creare una sorta di lobby a favore di una ridefinizione ideologica del software libero, evidenziandone cioè i vantaggi pratici per le aziende e coniando così il termine "Open Source". Ciò anche al fine di evitare l'equivoco dovuto al doppio significato di free nella lingua inglese, visto che spesso veniva interpretato come "gratuito" invece che come "libero" come già sottolineato in precedenza. Nel panorama del software libero si affiancano dunque due correnti di pensiero principali, quella di Stallman ampiamente approfondita e quella successiva dell'open source. Esse non sono fazioni contrapposte e oggigiorno collaborano a molti progetti concreti ma ci tengono a rimanere ben distinte dal punto di vista ideologico.

Riporto ora per completezza anche i principi chiave della filosofia dell'*Open Source Initiative* riassunti in 10 libertà fondamentali (e non più in quattro):

1. Libertà di redistribuzione (sta poi al singolo decidere se farlo gratuitamente o se far pagare il prodotto);
2. Libertà di consultare il codice sorgente;
3. Necessità di approvazione per i prodotti derivati;
4. Integrità del codice sorgente dell'autore;
5. Nessuna discriminazione verso singoli o gruppi di persone;
6. Nessuna discriminazione verso i settori di applicazione;
7. La licenza deve essere distribuibile;
8. La licenza non può essere specifica per un prodotto;
9. La licenza non può contaminare altri software;
10. La licenza deve essere tecnologicamente neutrale;

È evidente come si ritrovano le libertà del movimento del software libero, viene inoltre posta un attenzione specifica sull'aspetto delle licenze e verso i principi di non discriminazione che aprono la strada anche in ambito commerciale. Eric S. Raymond, uno dei fondatori del termine open source e della Open Source Initiative è accreditato sia dai principali hacker sia dai maggiori osservatori come colui che ha portato la missione dell'open source a Wall Street.

2.4.I vantaggi dell'Open Source

2.4.1 .Basso costo iniziale

L’adozione di software open source porta normalmente a un risparmio iniziale in termini di costi per licenze, ma anche di costi per gli aggiornamenti. I risparmi sono notevoli sugli aggiornamenti a fronte delle politiche commerciali di alcuni produttori di software proprietario, che propongono nuove versioni del propri pacchetti vantandole come rivoluzionarie rispetto alle versioni precedenti. In realtà, spesso le nuove versioni non portano vantaggi all'utente in termini di funzionalità aggiuntive, ma l'utente è portato ad acquistarle per incompatibilità con le versioni precedenti, o per non rimanere disallineato nei confronti di partner che usano la nuova versione. Un confronto economico più corretto deve essere però compiuto non solo sulla spesa iniziale, ma tra il TCO (total cost of ownership) delle soluzioni open source e il TCO delle soluzioni proprietarie. Oltre al costo delle licenze, nel TCO confluiscono le spese dei servizi di supporto, della formazione, i costi di migrazione, d'installazione e di gestione. Un noto argomento a favore degli OSS è che, con un minor costo delle licenze, il budget di

un'organizzazione può prevedere costi più alti per i servizi, e dunque in teoria più alti livelli di servizio.

2.4.2 .Indipendenza dai fornitori

Il modello open source impedisce il monopolio da parte dei produttori di software, e permette un maggiore controllo da parte del cliente, ove il software proprietario costituisce invece uno strumento di pressione (anche in sede di trattativa) da parte del venditore. Si pensi ad esempio alla correzione di problemi riscontrati su un pacchetto software. Nel caso open source, l'esistenza di comunità di sviluppatori diffuse nel mondo permette di ottenere rapidamente correzioni degli errori rilevati. Nel caso di software proprietario, si deve attendere che il produttore rilasci una patch, e nel frattempo non si può intervenire sul programma. Per ciò che riguarda il software sviluppato su richiesta, l'indipendenza dal fornitore consiste nel poter affidare il supporto di un prodotto open source a un azienda scelta dal cliente, laddove nel mondo del software proprietario solo il produttore, o un suo partner autorizzato, può supportare il proprio software. Oltre ai benefici della concorrenza, ciò consente anche di favorire imprese locali.

2.4.3 .Sicurezza

Disporre del codice sorgente dei programmi utilizzati all'interno della propria organizzazione permette un grado maggiore di sicurezza, specie in presenza di dati sensibili. Sono infatti più agevoli i controlli interni (ove nei software proprietari ci si deve affidare ai produttori) alla ricerca di eventuali *backdoor* o debolezze sfruttabili da attacchi esterni. Si cita come esempio il caso del DBMS Interbase di Borland. Dopo aver commercializzato per anni il prodotto, Borland decise di cedere i sorgenti alla comunità degli sviluppatori. Venne in breve riscontrato un difetto nel software, che era sempre sfuggito alla Borland, e che indeboliva la sicurezza delle procedure di accesso. Dunque la trasformazione del software in open source aveva reso possibile una maggiore sicurezza.

2.4.4 .Flessibilità

E' possibile realizzare versioni del kernel molto specializzate, il che porta facilmente a implementazioni particolari, ad esempio per dispositivi embedded, come vedremo nello specifico nel prossimo capitolo, o su hardware insufficiente per altri sistemi operativi. In generale, il software open source è più adatto ad essere personalizzato o esteso come funzionalità rispetto a un software proprietario.

2.4.5 .Interoperabilità

In termini di interoperabilità, il software open è più adatto del software proprietario. Lo scambio di dati e funzioni tra prodotti diversi implica difatti, in generale, la realizzazione di interfacce, e in caso di software proprietario solo chi detiene il codice sorgente può realizzare tali interfacce. Si noti anche, a proposito dell'interscambio di dati, che l'uso di un formato aperto per condividere documenti e/o file non è tecnicamente legato all'open source: software proprietari possono in teoria utilizzare comunque formati standard. Tuttavia, per i produttori di software proprietario, l'uso di formati chiusi è una politica commerciale che può portare vantaggi. Si pensi ai formati di Microsoft Office, che impongono di fatto uno standard a cui gli utenti devono adeguarsi, per cui ad esempio un utente è costretto ad acquistare la nuova versione di Excel (o di PowerPoint, o di Access) per poter collaborare con altri utenti che usano la nuova versione.

2.5.Diffusione

Per molti anni il modello OSS ha avuto diffusione soprattutto nel mondo degli sviluppatori e nel settore educational (università, enti di ricerca). E' più recente l'affermarsi di soluzioni open source anche presso utenti meno esperti di tecnologia, grazie alla nascita e alla diffusione dei distributori di OSS. A oggi è difficile stimare con precisione la diffusione del software open source: se infatti i produttori di software proprietario possono conteggiare il numero di licenze vendute (e, talvolta, stimare il numero di copie illegali in circolazione), nessuno può valutare il numero di installazioni di OSS, non essendoci tracce di acquisto. Neppure il numero di download dai siti web distributori di OSS è significativo, poiché non c'è una relazione rigorosa tra il numero di copie scaricate e quelle realmente installate e utilizzate. Secondo un articolo su BBC News [10], le tecnologie open source stanno diventando sempre più popolari, sia fuori che all'interno delle industrie IT. Il pezzo offre una panoramica di questo sviluppo e dei suoi effetti. Un punto interessante è il quesito sollevato da Paul Allen, editor di ComputerActive magazine, il quale si chiede perché il concetto di open source sia così sconosciuto alle masse, nonostante la diffusione di programmi che seguono questo modello. Per quanto riguarda Linux, il fattore principale è la frammentazione delle distribuzioni, che possono confondere e tenerne a distanza chi non è pratico. Guardando i numeri, abbiamo 60 milioni di download per OpenOffice.org, a partire da ottobre 2008: per fare un raffronto, basti pensare che gli utenti di Linux sono "solamente" 40-50 milioni! Inoltre, secondo Mozilla, si stimano 270 milioni di utilizzatori di Firefox, per non

parlare degli utenti di Thunderbird; non è errato dire che in certi contesti, tali prodotti hanno addirittura raggiunto Microsoft. La domanda che si è posto Allen ha in realtà una risposta di disarmante semplicità: la maggioranza della gente non è consapevole della filosofia Open Source, proprio perché usa questi prodotti senza interessarsi al movimento che c'è dietro, per poi magari sostenerlo. Allen osserva infine che, per creare equivalenti open delle più diffuse applicazioni commerciali, bisogna spesso ricorrere al *reverse engeneering*[1]: una critica non tanto velata al sistema proprietario, e alla mancata condivisione di certe specifiche.

2.5.1.In Italia

Forrester Research nel giugno del 2012 ha illustrato le nuove tendenze nell'open source alla quarta edizione della Open Source System Management Conference organizzata da Würth Phoenix a cui hanno partecipato 400 persone. Dai dati presentati emerge che in Italia oltre il 50% delle aziende è orientata all'utilizzo di soluzioni aperte e le valuta come "top technology goal", prima ancora del mobile o dei servizi cloud. Dai rilevamenti di Forrester risulta che la strategia informatica per le aziende in Nord America ed Europa, Italia compresa, si spinge verso l'introduzione di sistemi open source portando a un cambiamento culturale. Oltre il 40% delle aziende ritiene importante prendere in considerazione le soluzioni open per ridurre la dipendenza strategica dai fornitori software. La ragione principale dell'orientamento è stata inizialmente dettata dalla ricerca di un risparmio economico. Nel corso degli anni la tendenza ha generato un vero e proprio cambiamento a livello strategico culturale aziendale. Se in passato l'open source era prevalentemente un tema discusso in ambito tecnico, ora invece è sempre più il management a spingersi verso questa direzione. Nelle aziende intervistate da Forrester risulta, infatti, che il management possiederebbe addirittura una maggiore consapevolezza degli stessi sviluppatori sull'adozione di soluzioni open source ed è il principale sostenitore per l´introduzione di soluzioni open, con oltre il 50% per l'adozione di sistemi operativi o anche soluzioni di monitoraggio. A sostegno di queste tesi, riporta una nota di Wurth Phoenix, gli analisti hanno studiato alcuni casi tra cui anche quelli di realtà italiane come il gruppo Tecnica e Informatica Trentina, che grazie all'adozione di soluzioni open source per il monitoraggio informatico come NetEye, soluzione sviluppata in Italia, hanno ottenuto una riduzione del TCO, rispettivamente dell'80 e 90% [11].

1 L'ingegneria inversa (spesso si usa il termine retroingegneria) consiste nell'analisi dettagliata del funzionamento di un oggetto (dispositivo, componente elettrico, meccanismo, software, ecc.) al fine di costruire un nuovo dispositivo o programma che abbia un funzionamento analogo, magari migliorando od aumentando l'efficienza dello stesso, senza in realtà copiare niente dall'originale.

Anche nel campo della Pubblica Amministrazione si nota un continuo processo di adesione all'open source. Dai dati emersi dal consuntivo per le Pubbliche Amministrazioni Centrali (PAC), relativamente all'anno 2007, si pone in evidenza come l'80,39% delle PA ha dichiarato di utilizzare software open source all'interno dei propri sistemi informativi. Il dato mostra un aumento rispetto al precedente anno (2006), in cui tale valore è stato del 72%. Tra i principali vantaggi percepiti si pone al primo posto quello relativo al "risparmio economico" (68,29% degli utilizzatori), mentre al secondo si trova la "rapidità di acquisizione" (14,63%). Infine, è interessante notare come il 73,71% degli utilizzatori ha promosso sviluppo di soluzioni utilizzando software OS (dato in crescita rispetto al 65% del precedente anno). Interessanti le opportunità per gli operatori del mercato, infatti il 78,05% delle PAC utilizzatrici ha richiesto servizi relativi a componenti/prodotti OS alle aziende del settore. Tali dati sono sostanzialmente in linea con quelli rilevati dall'ISTAT presso le Pubbliche Amministrazioni Locali (PAL), dove tutte le regioni ed il 78,4% delle province ormai utilizza software OS [12].

2.6.I principali tipi di licenza

Nel seguente paragrafo verranno presentate le caratteristiche principali delle più importanti licenze che si possono trovare nell'ambiente del software Open Source.

2.6.1.Pubblico dominio

Il pubblico dominio (*public domain*) non è una licenza, perché non affonda i suoi diritti nel copyright. Il pubblico dominio, infatti, si riferisce a un'opera intellettuale che viene concessa a chiunque perché ne faccia ciò che vuole. Se riferiamo l'esempio a un software, apparentemente può sembrare che questo rappresenti il massimo della libertà, perché tutti possono fare tutto ciò che desiderano, ma in realtà non è esattamente così: chiunque può creare una versione modificata del software e rilasciarla con licenza proprietaria. In tal modo la libertà precedente viene persa, ma specialmente la conoscenza non viene diffusa.

2.6.2.Licenze FSF: GPL e LGPL

La GNU Public License (GPL), sviluppata da Richard Stallman insieme ad esperti legali, mira a perseguire gli obiettivi della FSF: chiunque possieda software GPL è libero di usarlo, copiarlo, modificarlo e ridistribuirlo (anche modificato), a patto di rendere sempre disponibile il codice sorgente e di non modificare la licenza del prodotto ridistribuito. Stallman inserì l'ultimo vincolo per impedire che qualcuno potesse approfittare del lavoro libero di altri per creare un software che poi avrebbe distribuito come software proprietario. Questa

particolarità della licenza è conosciuta come “effetto virale” della GPL, in quanto influenza tutti i prodotti derivati ed è, al contempo, la forza e la debolezza della licenza: non è possibile, infatti, inserire in un programma GPL del codice non libero (anche solo per scopo migliorativo), altrimenti il risultato non potrebbe essere il rilascio con licenza GPL con conseguente violazione del copyright. La GPL, a cui si è arrivati alla versione numero 3, ha dimostrato che ottimi programmi possono essere creati da una comunità di persone senza dover “chiudere” il codice sorgente. La GPL non impedisce che il software licenziato sia oggetto di business, ma pone il vincolo di mantenere lo stesso tipo di licenza. È possibile chiedere un corrispettivo per la distribuzione di un programma libero, e anche per eventuali personalizzazioni. Il software che ne deriva, però, deve essere necessariamente rilasciato con licenza GPL e deve essere disponibile il sorgente per il cliente. Esempi di prodotti rilasciati con licenza GPL sono il kernel Linux, The GIMP, il desktop manager GNOME, le librerie GTK, Moodle, Media Player Classic, Audacity, Inkscape, 7-ZIP, Videolan VLC Player, MySQL, il gestionale Compiere, SugarCRM, le distribuzioni OpenSuSE e Fedora, e via discorrendo.

La GNU Lesser General Public License (LGPL) è stata scritta nel 1991 (e aggiornata nel 1999) da Richard Stallman e Eben Moglen per superare un problema che, di fatto, rallentava l'utilizzo e la diffusione del software libero. In poche parole, la differenza tra la LGPL e la GPL è che la prima permette di inserire il codice licenziato sia in programmi liberi, sia in programmi proprietari senza che questo influisca sulla licenza finale del software. La licenza nacque inizialmente per permettere l'utilizzo di alcune librerie di uso comune anche in software proprietari, con l'obiettivo di diffondere maggiormente la conoscenza del software libero. La LGPL è assolutamente coerente col progetto GNU, infatti chiunque può, in ogni caso, licenziare il codice LGPL come GPL e includerlo, quindi, nei programmi GPL senza problemi. La LGPL è ritenuta un compromesso accettabile tra la GPL e altre licenze più permissive (ad esempio la BSD), e infatti, pur essendo nata per le librerie, viene usata anche da applicazioni come OpenOffice. Dal punto di vista degli sviluppatori, il vantaggio è quello di poter usare del codice senza che questo influisca sulla licenza finale del prodotto che ne deriva. Le modifiche apportate al codice LGPL, comunque, vanno rese pubbliche contribuendo, quindi, al miglioramento generale del software.

2.6.3.Licenze Open Source: MPL e BSD License

La Mozilla Public License (MPL) e la Berkeley Software Distribution (BSD) License non sono considerate licenze libere secondo la Free Software Foundation, ma rientrano nella Open

Source Definition secondo la OSI. La MPL caratterizza i prodotti derivati dal progetto Mozilla. Rispetto alla GPL, la MPL introduce delle clausole che permettono di usare il codice in maniera diversa rispetto a quanto bisogna fare con i prodotti licenziati GPL. In particolare, le differenze sono due: un software MPL può essere miscelato con software commerciale, e le modifiche possono essere mantenute private e non restituite all'autore originale. Per evitare problemi, alcuni programmi MPL sono rilasciati anche con licenza GPL e LGPL in modo che possano essere inseriti in software GPL senza andare contro la licenza della FSF. Esempi di programmi con licenza MPL sono Firefox, Thunderbird e tutta la suite Mozilla.
La BSD è molto simile nei principi alla MPL, dalla quale si differenzia per alcune piccole clausole. La BSD, nata originariamente presso l'università di Berkeley ai tempi dello sviluppo dei primi Unix liberi, permette la redistribuzione e l'uso del software licenziato con essa, sia in forma di codice sorgente che in forma di codice eseguibile. Non impone il rilascio pubblico del codice sorgente delle opere derivate, avvicinandosi così molto al concetto di "pubblico dominio". Uniche condizioni sono la citazione delle note di copyright, delle condizioni della licenza e della clausola che solleva l'autore originario dall'obbligo di qualunque garanzia. Nel sito della OSI esistono molte altre decine di licenze considerate Open Source, come la Apache Software License o la Sun Public License. La licenza viene scritta a seguito di particolari esigenze dell'autore del codice, e spesso le differenze tra una e l'altra non sono molto facili da cogliere. Tendenzialmente, comunque, tutte fanno riferimento nei loro principi generali alle tipologie appena descritte (GPL, LGPL e MPL/ BSD).

2.6.4.Dual Licensing

Generalmente, ogni software viene rilasciato con un solo tipo di licenza ma nulla impedisce all'autore, possessore dei diritti, di rilasciare il suo lavoro contemporaneamente sotto diversi tipi di licenza (normalmente non più di due). Questo particolare esercizio del copyright prende i l nome di *Dual Licensing*. Il *Dual Licensing* è strettamente correlato al software Open Source, e il suo utilizzo può essere di volta in volta apprezzato o odiato dai sostenitori del software libero. La reazione positiva si ha quando la seconda licenza serve a rendere il programma compatibile con le richieste di altre licenze libere. È il caso ad esempio del browser Firefox, il cui codice è rilasciato con licenza MPL ma anche GPL, in modo che possa essere inserito anche in software GPL senza violare i diritti previsti da quest'ultima licenza. Spesso, però, accade anche questo: l'autore mette a disposizione il suo software distribuendolo con una licenza libera (spesso la GPL) ma anche, a richiesta, con una licenza commerciale. La differenza è chiara: il programmatore che usa il prodotto libero non potrà

"chiudere" il risultato del suo lavoro, mentre chi sta utilizzando il prodotto licenziato con licenza commerciale potrà fare a meno di distribuire il codice alla comunità e al cliente finale. In questo caso vi sono delle perplessità da parte del mondo del software libero, ma è anche vero che il codice viene comunque rilasciato come Open Source e, quindi, può essere usato da tutti. Esempi di software rilasciati con la doppia licenza libera/proprietaria sono il famoso database MySQL e le librerie grafiche Qt di Trolltech che stanno alla base del desktop manager KDE di Linux. Per il vero "proprietario" del software il rischio del *Dual Licensing* è quello di far perdere fiducia alla comunità di sviluppatori, che potrebbe quindi rivolgersi verso altri prodotti totalmente liberi abbandonando il proprio prodotto, con i conseguenti aumenti di spesa per lo sviluppo.

CAPITOLO 3

3.Possibilità Open Source per i Sistemi Embedded

3.1.Sommario

In un contesto così nuovo ed in continuo mutamento come quello dei sistemi embedded l'approccio Open Source si rivela vincente. Nel corso di questo capitolo analizzeremo le motivazioni principali e forniremo alcuni esempi. Se dovessimo identificare tre pilastri dello sviluppo dei sistemi embedded allora potrebbero essere i seguenti tre principi: l'integrazione, l'innovazione e l'industrializzazione; come riporta il paper: *Embedded system design with open source software: doing it right* di *Knut Yrvin*[2] *and Burkhard Stubert*[3] [13]. Queste non sono regole scolpite nella pietra e nemmeno standard di fatto, sono invece le sfide comunemente condivise dagli addetti del settore che lo sviluppo in questo contesto porta ad affrontare.

3.1.1.Integrazione

Il coordinamento con i partner esterni e gli sviluppatori software saranno inevitabilmente parte dell'equazione totale del progetto. In questo ambiente di sviluppo altamente esigente, per gestire in modo efficace il processo, una “pianificazione strategica” deve essere effettuata per l'esecuzione in parallelo degli sforzi del team di sviluppo. Una strategia Open Source aiuta in questo caso, per la sua stessa natura di essere fondata su un intero ecosistema di sviluppatori attorno ad una comunità. Questo insieme di individui, socialmente e tecnologicamente integrato, condivide una comune filosofia di interscambio e crescita reciproca. In questo ambiente, l'integrazione è naturalmente più fluida. Non è raro, ad esempio, prendere dei blocchi di codice già esistente da una comunità ed adattarli per il proprio progetto, chiedendo agli sviluppatori di contribuire. Dopo tutto una comunità Open Source è, in qualsiasi momento, alla ricerca della prossima direzione verso cui incanalare la propria creatività.

3.1.2.Innovazione

Quando si cerca di progettare un nuovo prodotto innovativo, nel settore embedded, solitamente, non si possiedono in casa tutte le conoscenze necessarie per farlo. Anche in questo caso risulta evidente come un approccio Open Source possa aiutare. La nuova

2 Community Manager for Qt Development Frameworks
3 Sales Engineer for Nokia Qt Development Frameworks at Nokia

possibilità permette di attingere a conoscenze detenute al di fuori dell'azienda. Un pessimista potrebbe pensare che questa tecnica è controproducente, dato che così le conoscenze necessarie non saranno mai un patrimonio dell'azienda. In realtà non ci sono prove che questo sia un problema reale; in pratica gli sviluppatori di un'azienda utilizzano esempi di codice o altre componenti Open Source, li adattano alla propria esigenza e li osservano nei minimi particolari imparando le conoscenze necessarie da questi.

3.1.3.Industrializzazione

Il software embedded, generalmente, viene venduto in modo industriale su larga scala. Progettare il software adeguato che sia scalabile[4] e quindi adatto a questa sfida non è una cosa banale, l'introduzione di test di interoperabilità[5] deve essere parte del processo. Quando lo sviluppo open source viene eseguito con diligenza e con un adeguato livello di pianificazione strategica, non vi è alcuna differenza per la sua scalabilità rispetto a quella di uno sviluppo proprietario su larga scala. Se non altro, l'interoperabilità è probabile che sia una sfida meno impegnativa rispetto al provisioning[6] che dovrà essere effettuato per effettuare tutti i test all'interno di un sistema proprietario. Soprattutto in un progetto su larga scala, nessun numero di test è mai troppo elevato.

3.2.Panorama

Quali sono le alternative software Open Source disponibili per i sistemi embedded? La risposta a questa domanda è tratta dal paper *Open Source in Embedded System Development* di *Jeremy Bennett,* esperto della *Embecosm; s*i sviluppa nel prossimo paragrafo divisa in settori di interesse e raccoglie gli esempi più diffusi. Altre informazioni sui software Open Source in campo embedded si possono trovare su Wikipedia (www.wikipedia.org) oppure su portali dedicati alle iniziative Open come SourceForge (sourceforge.net) e GitHub (github.com).

3.2.1.Hardware modeling

Con la nascita della modellazione hardware i software Electronic Design Automation (EDA)

4 Il termine scalabilità, nelle telecomunicazioni, nell'ingegneria del software, in informatica, e in altre discipline, si riferisce, in termini generali, alla capacità di un sistema di "crescere" o "decrescere" (aumentare o diminuire di scala) in funzione delle necessità e delle disponibilità.

5 Interoperabilità è in ambito informatico, la capacità di un sistema o di un prodotto informatico di cooperare e di scambiare informazioni o servizi con altri sistemi o prodotti in maniera più o meno completa e priva di errori, con affidabilità e con ottimizzazione delle risorse.

6 Processo mediante il quale un amministratore di sistema procede alla preparazione di risorse e privilegi, in modo da garantire il rilascio di un nuovo servizio. Inteso qui come “attrezzaggio” per poter effettuare i test.

sono stati una roccaforte del software proprietario. Alcune soluzioni Open Source si sono comunque affermate: per la modellazione di alto livello *SystemC* ha sempre avuto un'implementazione open e per la modellazione accurata e la traduzione del linguaggio *Verilog Hardware Description Language (HDL*), *Verilator*, nella sua terza revisione, è molto stabile e robusto. Così come sono diffusi *Icarus Verilog*, sempre per il linguaggio Verilog, e *GHDL* per *VHDL*. In questo settore il software proprietario risulta più veloce è affidabile delle alternative open che comunque sono molto valide per la maggior parte delle applicazioni. Più in generale il *Free Electronics Lab* (formalmente il Fedora Electronics Lab) contiene una collezione di strumenti EDA Open Source che sono entrati a far parte delle distribuzioni standard di Fedora Linux.

3.2.2.Compiler tool chains

Per quanto riguarda i compilatori, il *GNU Compiler Collection* (GCC) ha dominato il settore negli ultimi 25 anni, supportando circa 40 architetture nella sua distribuzione standard, con un ampio range di linguaggi (C, C++, Java, ObjectiveC/C++, Fortran and Ada). In ogni caso anche *GCC* sta mostrando la sua età e *LLVM*, con una architettura più moderna, sta diventando una alternativa molto popolare. Attualmente solo C e C++ sono ben supportati anche se sta crescendo l'interesse delle aziende per questo compilatore. Apple e ARM's assicurano che LLVM continuerà ad avanzare rapidamente, ciò è dimostrato ad esempio dalla diffusione di questo compilatore in Xcode [14]. *GCC* e *LLVM* in realtà non sono ottimizzati per piccoli processori spesso diffusi nei sistemi embedded, per il supporto al C il *Small Device C Compiler* (*SDCC*) è una buona alternativa. Il linguaggio Java è approdato in ritardo in campo Open Source e il *GCC Java compiler* generalmente soffre di questioni di compatibilità. Fortunatamente Sun ha reso disponibile il sistema *OpenJDK* . Esso non è completo come la sua controparte commerciale ma rende Java disponibile per lo sviluppo nel campo dell'Open Source.

3.2.3.Libraries

Tutto il software viene utilizzato attraverso librerie, qui verranno considerate solo quelle principali relative a C/C++. Queste librerie sono caratterizzate dal fatto di avere generalmente delle licenze “non virali” e più permissive. Per piccoli sistemi embedded il pacchetto GNU è spesso usato con *newlib*, una libreria molto leggera disponibile anche per sistemi Real Time Operating Systems (RTOS). La libreria principale di GNU è *glibc*, ma troppo estesa per la maggior parte delle applicazioni embedded che sono contraddistinte da

poca memoria a bordo e limitate e precise funzionalità. Un buon compromesso è rappresentato da *uClibc*, che è abbastanza ricca da essere usata anche con Linux.

3.2.4.Graphical Computer-aided Software Engineering (CASE)

GCC, LLVM e SDCC sono strumenti da linea di comando cioè che non nascono con un interfaccia grafica. Il principale ambiente grafico di sviluppo è *Eclipse*. La sua configurabilità e la licenza permissiva con cui è rilasciato hanno contribuito a renderlo molto personalizzabile e adatto ad un alto numero di compiti. Sia GCC che LLVM hanno un'ottima integrazione con *Eclipse* che rimane il software più largamente diffuso in questo settore.

3.2.5.Source code debugging

Per l'analisi degli errori del codice sorgente, *GNU Debugger* (*GDB*) è stato da sempre il principale strumento Open Source, ed è anche ben integrato in *Eclipse*. Il progetto LLVM sta lavorando su un suo proprio debugger (*LLDB*), ma ancora non ha raggiunto la maturità necessaria da essere paragonato a *GDB*.

3.2.6.Version control

Per il tracciamento delle revisioni e per tenere sotto controllo gli aggiornamenti da una versione ad un altra c'è una vasta scelta. *CVS* e *Subversion* sono entrambi diffusi. Per quanto riguarda il mantenimento delle repositories, *git* è il più conosciuto ed utilizzato dato che fa parte delle varie “distro” di Linux. Citiamo anche *mercurial* e *bazaar*.

3.2.7.Build systems

Lo strumento più longevo e diffuso è senz'altro *make*, una componente standard di ogni distribuzione Linux. Ci sono molte implementazioni differenti ma la versione originale di GNU è la più comune e potente. In GNU ci sono alcuni “autotools” (autoconf, automake e libtool) affermati che si basano su *make* ma necessitano di alcuni accorgimenti nell'uso. Una nuova alternativa è *cmake*. Per i sistemi Java, *Ant* è comunemente diffuso.

3.2.8.Operating systems

Molti sistemi operativi Real-Time sono Open Source o comunque offrono varianti Open Source della versione proprietaria. Possiamo citare ad esempio *RTEMS*, *FreeRTOS* e *eCos*. In molti casi gli stessi distributori di questi prodotti offrono la versione “chiusa” che include supporto tecnico e alcune garanzie, oppure cartificazioni per usi specifici. L'uso di *Linux* è

emblematico nel campo Open Source ed è largamente impiegato in campo embedded. Alcune aziende offrono versioni commerciali di *Linux* per sistemi embedded, infatti se il kernel è Open Source non c'è nessuna ragione che ostacola le aziende ad aggiungere pacchetti proprietari per specifiche operazioni. Per sistemi embedded inoltre *BusyBox* si usa come una shell leggera da far girare su di un kernel (ad esempio nei routers Netgear). *Android* è un derivato di Linux e, anche se Google è stata criticata per tenere alcune parti sotto licenza proprietaria, la maggior parte di esso è Open Source.

3.2.9.Middleware and applications

Il numero di applicativi è enorme quindi poniamo l'attenzione solo su quelli che possono ritenersi interessanti per i sistemi embedded. Per la sincronizzazione tra dispositivi, *NTP* è sempre stata una buona alternativa Open Source. Per il database le maggiori opzioni Open Source sono *MySQL* e *PostgreSQL*. Molti sistemi embedded necessitano di un web server e in questo campo *Apache* viene usato nel 70% di tutti i web server nel mondo.

Ora si riporta la tabella riassuntiva di quanto espresso finora con i collegamenti alle pagine ufficiali:

Tools	Source
Hardware modeling SystemC Verilator Icarus Verilog GHDL Free Electronics Lab	 www.systemc.org www.veripool.org/wiki/verilator iverilog.icarus.com ghdl.free.fr spins.fedoraproject.org/fel
Compiler tool chains GNU Compiler Collection (GCC) LLVM Small Device C Compiler (SDCC) OpenJDK	 gcc.gnu.org llvm.org sdcc.sourceforge.net openjdk.java.net
System C/C++ libraries newlib glibc uClibc STLport C++	 sourceware.org/newlib www.gnu.org/s/libc uclibc.org www.stlport.org
Graphical CASE Eclipse	 www.eclipse.org
Source code debugging The GNU Debugger (GDB) The LLVM Debugger (LLDB)	 www.gnu.org/s/gdb lldb.llvm.org

Tools	Source
Version control	
CVS	www.nongnu.org/cvs
Subversion	subversion.tigris.org
Git	git-scm.com
Mercurial	mercurial.selenic.com
Bazaar	bazaar.canonical.com/en
Build systems	
GNU make	www.gnu.org/software/make
GNU autoconf	www.gnu.org/s/autoconf
GNU automake	www.gnu.org/software/automake
GNU libtool	www.gnu.org/software/libtool
cmake	www.cmake.org
Ant	ant.apache.org
Operating systems	
RTEMS	www.rtems.com
FreeRTOS	www.freertos.org
eCos	ecos.sourceware.org
Linux	www.kernel.org
BusyBox	www.busybox.net
Android	www.android.com
Middleware and applications	
NTP	www.ntp.org
MySQL	www.mysql.com
PostgreSQL	www.postgresql.org
Apache	www.apache.org

3.3.Diffusione

Rivolgendo un occhio di riguardo ai Sistemi Operativi si illustra ora il risultato del sondaggio effettuato da UBM Tech Electronics[7] *2013 Embedded Market Study* [15], in cui viene presentato un rapporto completo sulla diffusione degli OS tra gli sviluppatori embedded. I dati di questo sondaggio vanno presi con la dovuto accortezza, va tenuto in considerazione infatti il profondo mutamento che contraddistingue questo settore, in cui ogni mese ci sono importanti novità. I dati raccolti sono presi a partire da un campione di 2098 intervistati, tutti professionisti del settore Embedded con un affidabilità del 95% +/- 2,13%, secondo UBM.
È notevole il dato generale che vede i Linux-based OS, Android compreso, raggiungere il 50% delle preferenze. Se togliamo Android invece, che registra la più alta crescita personale di ogni altra piattaforma, e vale il 16% atteniamo che Linux riceve il 34% delle preferenze. Sono significativi i grafici del sondaggio che riporto e ritengo più esaustivi di molte parole.

7 http://tech.ubm.com/

Illustrazione 5: Diffusione attuale degli OS nei sistemi embedded

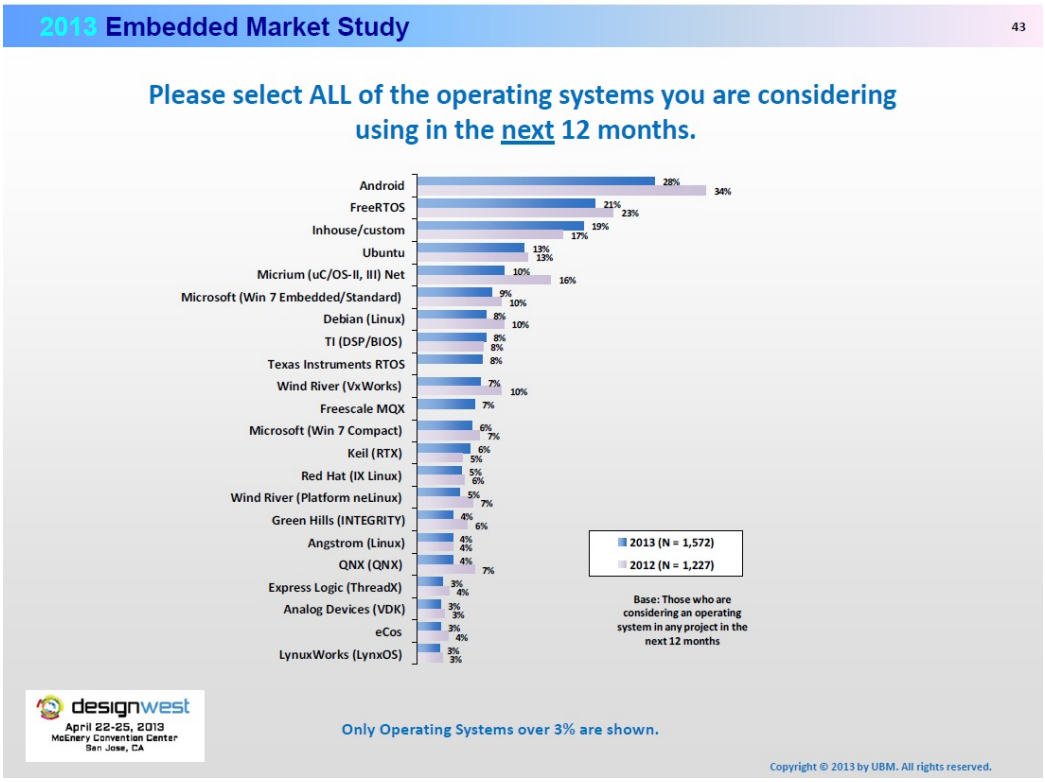


Illustrazione 6: Situazione futura degli OS nei sistemi embedded

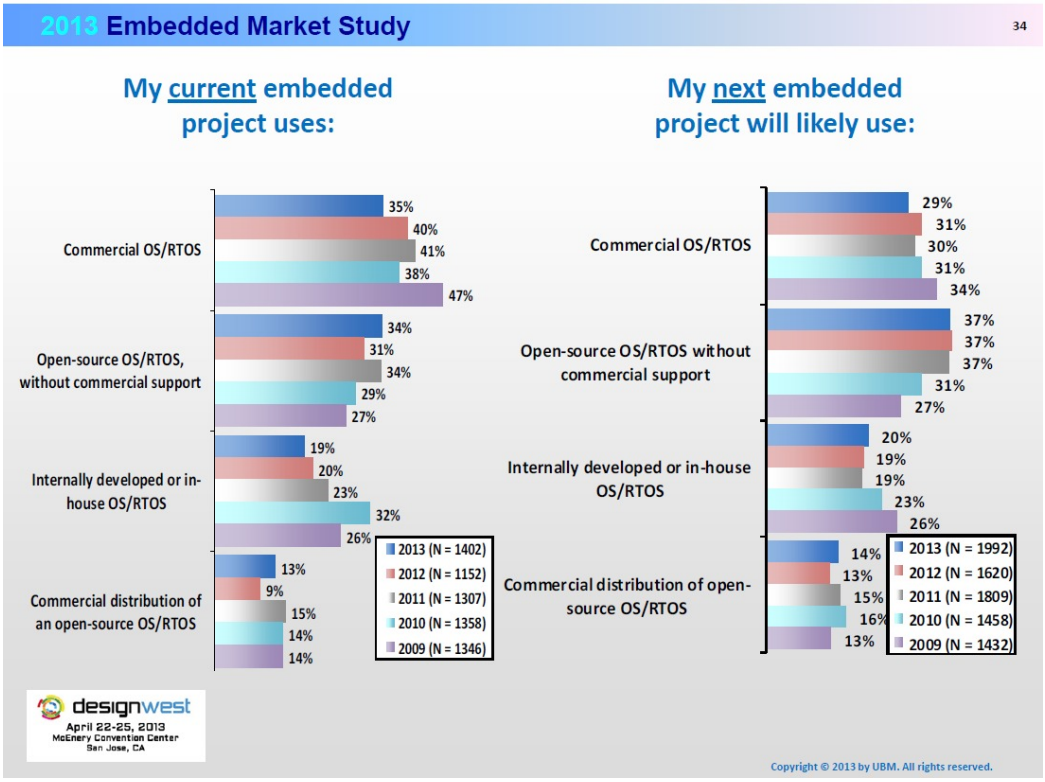


Illustrazione 7: Che tipologia di OS viene impiegata

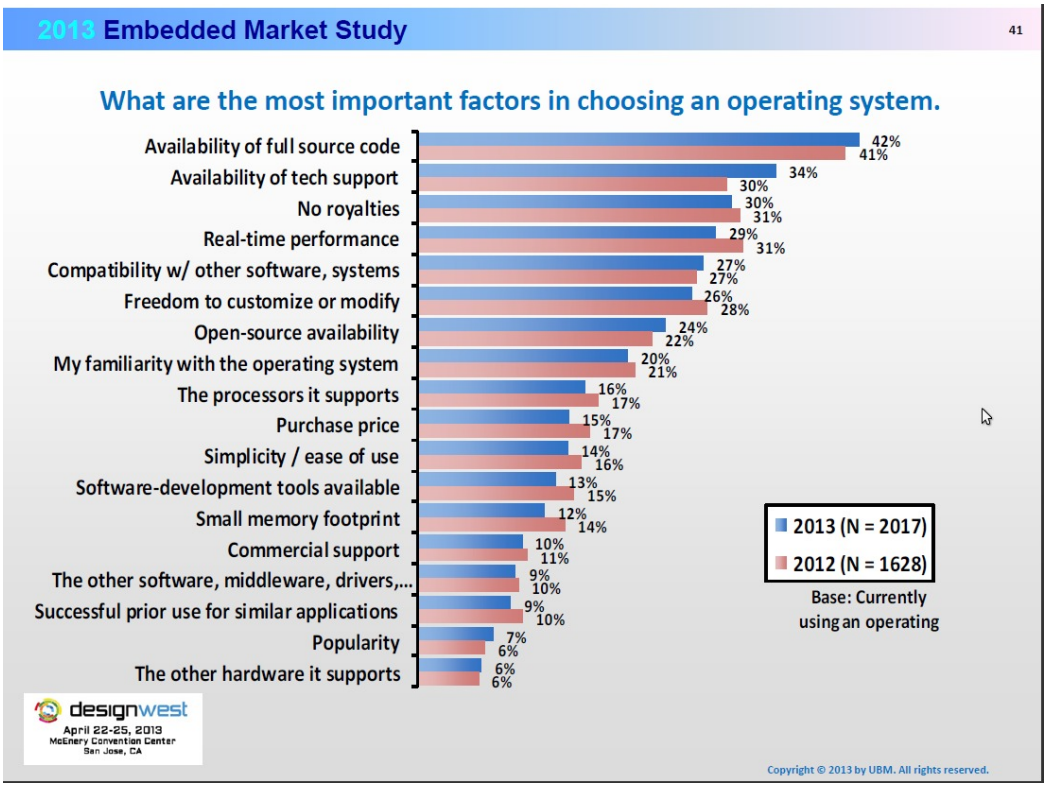


Illustrazione 8: Fattori per la scelta di un OS

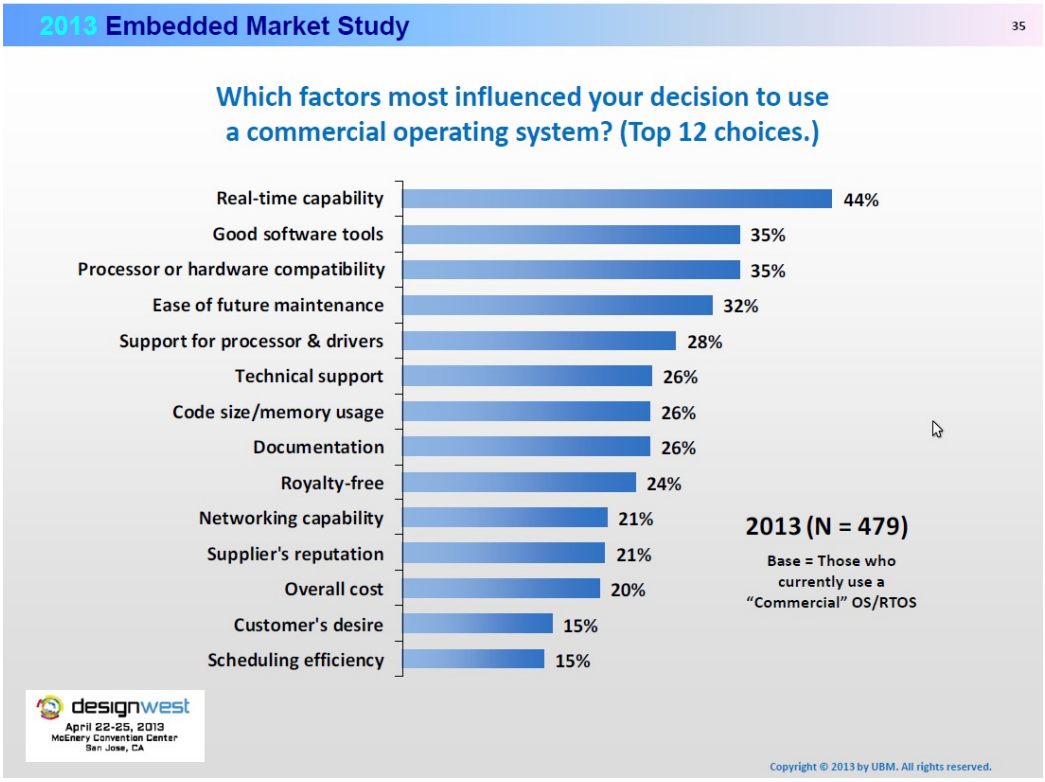


Illustrazione 9: Perché un sistema Proprietario

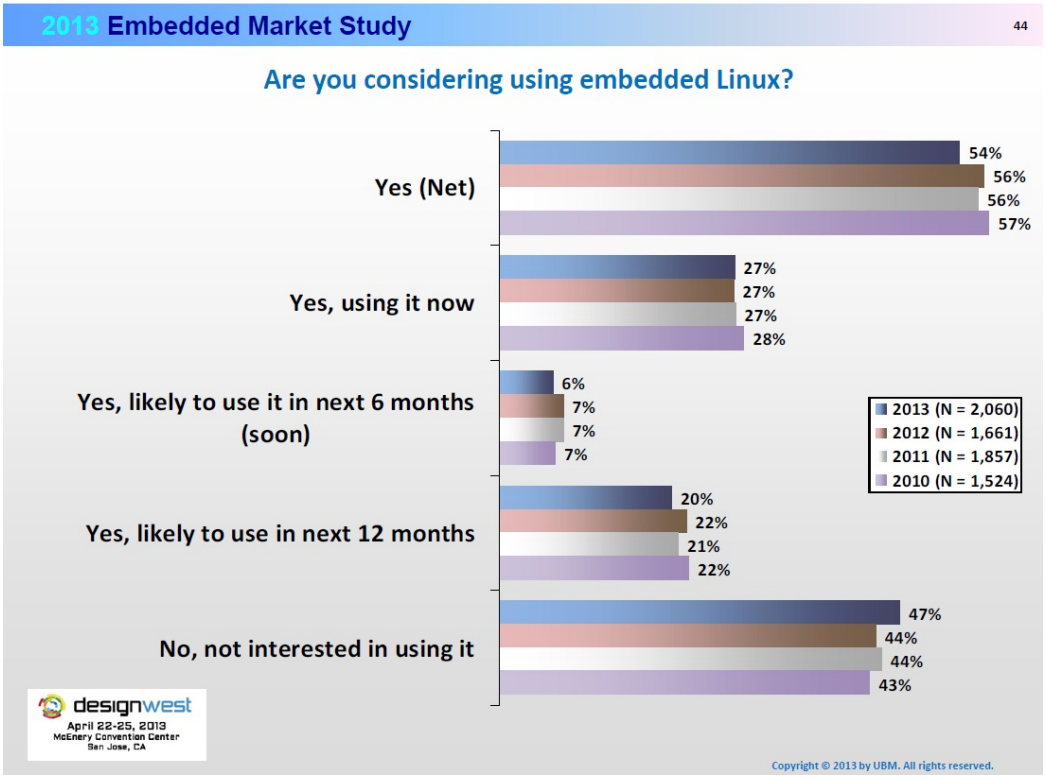


Illustrazione 10: Embedded Linux è nei programmi dei pofessionisti?

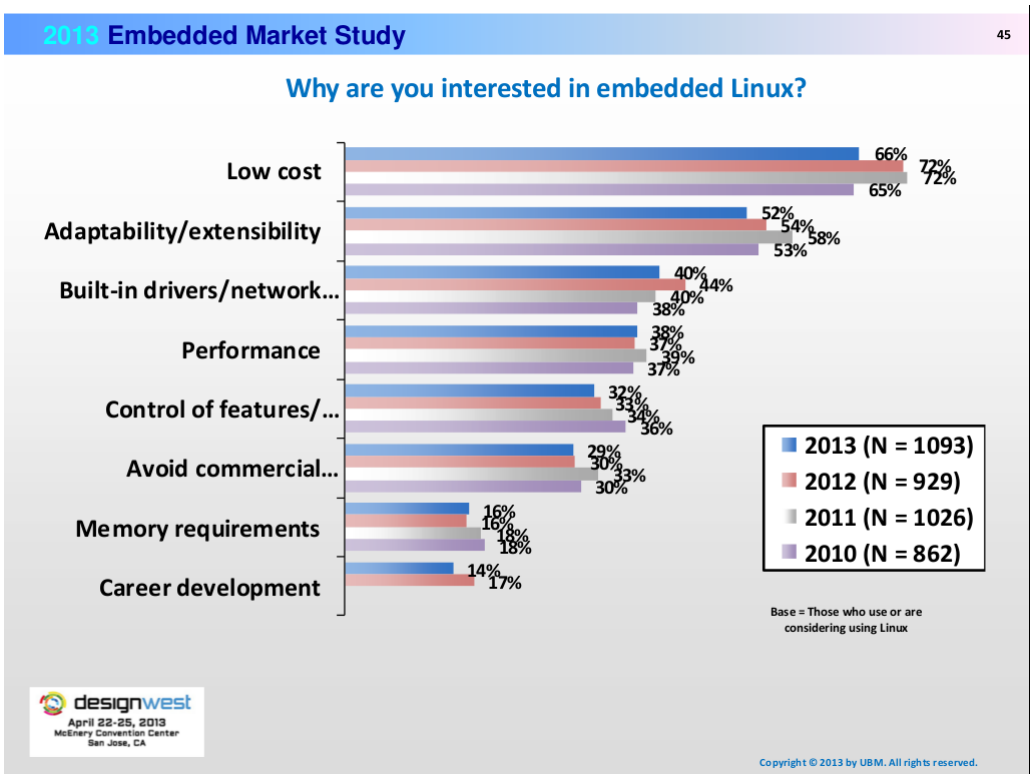


Illustrazione 11: Perché scegliere Embedded Linux

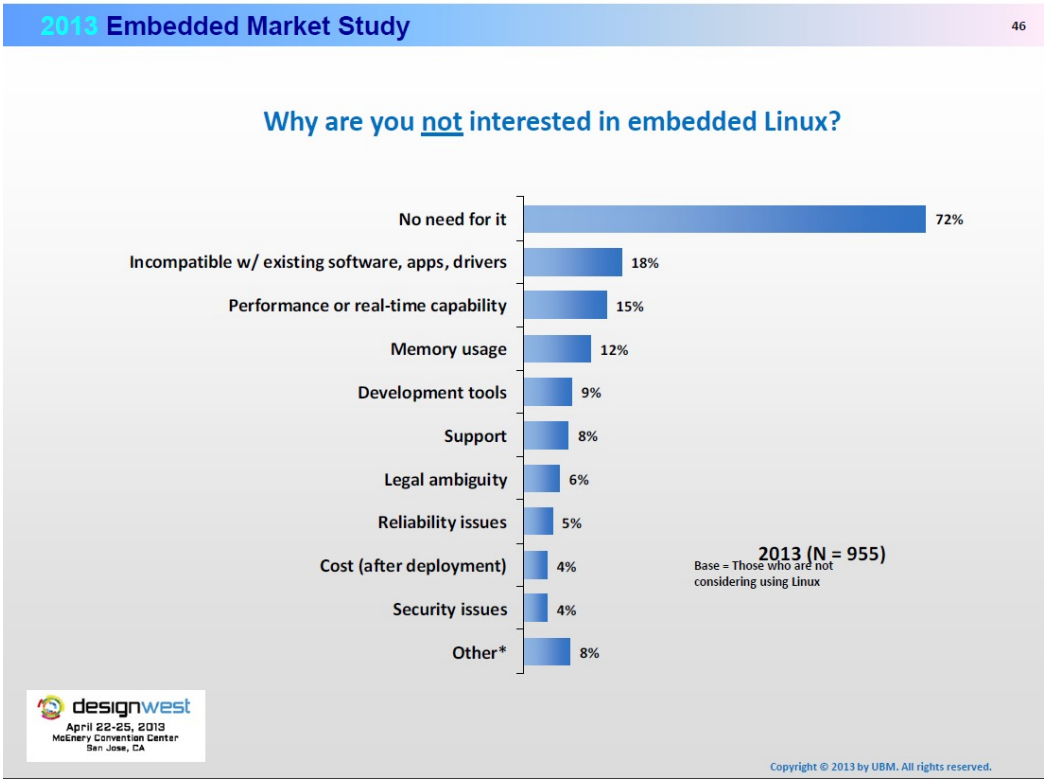


Illustrazione 12: Perché non sceglierlo

Riassumendo i risultati del sondaggio possiamo affermare che Linux (considerando tutte le distribuzioni presenti e sommandole come in tabella 1) rappresenta il più diffuso sistema operativo, superando le soluzioni “in-house/custom” che raggiungono il 24% (Illustrazione 5). D'altro canto se consideriamo insieme tutte le altre piattaforme, per la maggior parte proprietarie, di Real Time OS (RTOS), senza contare gli OS embedded di Microsoft e di Texas Instruments (DSP platform) si raggiunge il 57% delle preferenze (Illustrazione 5). Quindi i sistemi prettamente RT ancora hanno un ruolo fondamentale sulla scena[8] (Illustrazione 9). Considerando proprio gli RTOS si evidenzia che cinque di questi registrano perdite negli anni, tre rimangono invariati e solo due mostrano di crescere: FreeRTOS e VDK di Analog Devices. Nel mentre i sistemi di Microsoft mantengono un andamento costante.
Per quanto riguarda la percezione di cosa sarà il futuro, Android, con il 28%, rappresenta la percentuale più alta tra i sistemi che ci si aspetta di adottare (Illustrazione 6). Ma è anche mostrato come al 34% di quelli che si aspettavano di usare Android un anno fa sia poi seguito un effettivo utilizzo ad oggi del 16%. Ciò dimostra che c'è una discordanza tra le aspettative forse influenzate dalla foga del periodo e poi l'effettiva applicabilità di certe soluzioni. Cali simili di un paio di punti percentuali si registrano anche per quanto riguarda Linux, FreeRTOS e altri RTOS che perdono anche di più.

	2013	**2012**
Android	16%	13%
Ubuntu	13%	12%
Debian	8%	8%
Red Hat	5%	5%
Wind River	4%	4%
Angstrom	4%	4%
TOTAL LINUX	**50%**	**46%**

Tabella 1: I veri vincitori del sondaggio sono i sistemi Linux-based

3.4.Vantaggi dei FOSS nei Sistemi Embedded

I vantaggi e gli svantaggi del software Open Source per i sistemi embedded ricalcano quasi completamente quelli visti per il solo software Open Source nel paragrafo 2.4. In questo paragrafo si vuole invece dividere i vantaggi in parte già illustrati negli ambiti di interesse

8 Si ricorda che Linux non è un sistema prettamente Real Time e verranno evidenziati nel capitolo relativo a Linux Embedded i suoi vincoli sul Real Time e come vengono risolti.

relativi al software per sistemi embedded [16].

3.4.1.Vantaggi per gli sviluppatori

Nel campo Open Source l'innovazione è continua dato che è continuo l'impegno delle comunità. Questo fornisce un valore aggiunto in quanto si può trovare sempre codice aggiornato per le esigenze più disparate. Questo non vuol dire che le conoscenze diventino obsolete, anzi, le capacità acquisite non sono legate al particolare OS ma possono venire impiegate anche se si cambia sistema o azienda. La qualità del software è elevata in quanto c'è controllo costante da parte della comunità sul software e sono molti gli occhi pronti a scovare i bug. C'è larga scelta di prodotti e la facilità di sviluppo grazie al supporto disponibile è garantita. Importante è tenere in considerazione il basso costo, o addirittura nullo, per la maggior parte delle alternative.

3.4.2.Vantaggi per i manager

É di fondamentale importanza mantenere una certa indipendenza dalle software house per non essere soggetti ad imposizioni di prezzo, aggiornamenti obbligati e scelte tecniche che magari non condividiamo. Questo è garantito solo quando si usa software Open Source. Quest'indipendenza si traduce anche nell'effettiva proprietà del prodotto messo sul mercato per non dovere royalties al di fuori. Con la possibilità di riusare il software si ha la possibilità di diminuire i tempi di immissione sul mercato di un nuovo prodotto. Posso fare cose nuove partendo da software che riadatto senza spreco di tempo per ricominciare da zero.

3.4.3.Vantaggi per le compagnie

Tutti i vantaggi citati per sviluppatori e manager si applicano anche qui. Inoltre sviluppare con software libero può attrarre l'attenzione di sviluppatori talentuosi delle comunità Free and Open Source Software (FOSS).

3.4.4.Vantaggi per i clienti e gli utilizzatori

La presenza di formati aperti garantisce la proprietà effettiva dei propri dati con la possibilità di utilizzarli per tutto il tempo necessario, molto utile quando si cambia l'hardware ad esempio. La qualità e l'affidabilità è garantita dal grande numero di persone che usano questi strumenti anche su dispositivi diversi. La sicurezza e la privacy sono sotto una più stretta sorveglianza, quando parliamo di dati sensibili il codice è vagliato da molti esperti.

3.5.Difetti

Linux è un sistema operativo general purpose che supporta un gran numero di piattaforme, e per quanto possa essere affinato e personalizzato non sarà mai paragonabile ad un sistema ad hoc finemente tarato per uno specifico hardware. Richiede quindi un forte lavoro di personalizzazione per poterlo utilizzare con utilità sul sistema embedded che vogliamo. Inoltre ci sono ancora delle incertezze di tipo legale sulle licenze Open Source, per esempio la GPL non è applicabile in tutti i paesi. Bisogna soprattutto saper prestare molta attenzione quando si mischiano sorgenti con licenze diverse e documentarsi con cautela sulle leggi in materia dei singoli stati.

3.6.Kernel per Sistemi Embedded

3.6.1.Tipi di Kernel

Il kernel costituisce il nucleo di un sistema operativo, il suo compito è quello assicurare ai processi in esecuzione sull'elaboratore un accesso a tutte le risorse hardware. Dato che i processi in esecuzione sono sempre più di uno è il kernel che ha la responsabilità di gestire, tra le altre cose, lo *scheduling*. Le funzioni principali di un Kernel sono:

- fornire un interfaccia comune con la quale le applicazioni interagiscono con il resto del sistema attraverso un insieme di funzioni indicate come *chiamate di sistema* (*system calls*);
- consentire l'accesso ad un'ampia varietà dispositivi e alle loro peculiari funzionalità attraverso l'integrazione di driver specifici e alle disponibilità di meccanismi di comunicazione fra questi ed il sistema operativo;
- regolare, attraverso una serie di servizi e sottosistemi, le modalità di esecuzione delle applicazioni, le comunicazioni inter-processo e la distribuzione delle risorse del sistema.

L'accesso diretto all'hardware può essere anche molto complesso, quindi i kernel usualmente implementano uno o più tipi di astrazione, il cosidetto *Hardware abstraction layer*. In base al tipo di astrazione è possibile individuare una classificazione per i vari kernel [17]. Esistono essenzialmente quattro tipi di astrazione:

- Kernel monolitici: implementano in modo diretto un layer di astrazione completo dell'hardware;
- Microkernel: forniscono un limitato livello di astrazione dell'hardware, si appoggiano su software di tipo device driver o server per espandere le proprie funzionalità;

- Kernel ibridi: simili all'approccio a microkernel ma implementano direttamente alcune funzioni aggiuntive al fine di incrementare le prestazioni globali del sistema;
- Esokernel: rimuovono tutte le complicazioni legate all'astrazione dell'hardware e si limitano a garantire l'accesso concorrente allo stesso, permettendo alle singole applicazioni di implementare autonomamente le tradizionali astrazioni del sistema operativo per mezzo di speciali librerie.

Ogni tipo di astrazione presenta rispetto alle altre vantaggi e svantaggi.

3.6.2.Kernel Monolitici

Un kernel monolitico presenta un'interfaccia virtuale di altro livello sull'hardware consentendo ai processi sovrastanti di interfacciarsi con esso tramite delle chiamate di sistema. Le *system calls*, appunto, implementano una serie di funzioni fondamentali per il funzionamento della macchina. Ad esempio la gestione di più processi (*multitasking*) e la gestione della memoria.
I servizi di base sono gestiti attraverso moduli che lavorano in modalità supervisore anche detta modalità kernel.
Il più grande degli svantaggi di un kernel di tipo monolitico è l'impossibilità di aggiungere un nuovo dispositivo hardware senza implementare il relativo modulo al kernel; tale operazione inoltre richiede la ricompilazione dello stesso kernel. In alternativa è possibile già pre-compilare un kernel con tutti i moduli di supporto all'hardware, ma il prezzo da pagare è quello di dover aumentarne la dimensione anche di molti megabyte. Per aggirare parzialmente questo problema i moderni kernel monolitici, ad esempio il kernel Linux e FreeBSD hanno la possibilità di caricare i moduli in fase di esecuzione a patto che questi siano già previsti in fase di compilazione. Questa soluzione ha permesso di ottenere oggigiorno dei kernel efficienti e molto flessibili. La figura qui sotto riportata evidenzia molto semplicemente l'astrazione del kernel monolitico e la sua semplicità di interpretazione.

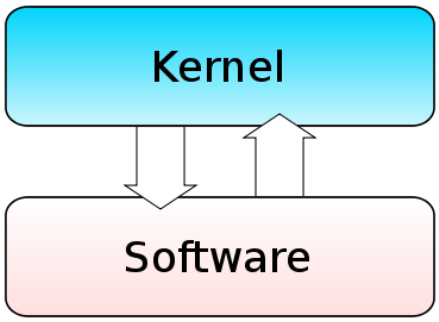


Illustrazione 13: rappresentazione grafica di un kernel monolitico

*3.6.3.*Microkernel

La caratteristica del microkernel è quella di definire diverse macchine virtuali (servers) semplici, al di sopra dell'hardware. Un insieme di *system calls* sono utilizzate per implementare i servizi di base del sistema operativo, dividendoli dai servizi di più alto livello; di modo che anche se si dovessero avere dei problemi con i moduli di alto livello questi non influenzano l'intero sistema. I servizi di base sono generalmente thread handler, spazi di indirizzamento e IPC (inter-process comunication). Uno svantaggio di questo tipo di kernel è un calo delle prestazioni rispetto ad un kernel monolitico. I microkernel sono spesso usati in sistemi embedded, in applicazioni mission critical di automazione robotica o di medicina, a causa del fatto che i componenti del sistema risiedono in aree di memoria separate, private e protette. Ciò non è possibile con i kernel monolitici, nemmeno con i moderni moduli caricabili. La figura seguente rappresenta l'approccio a microkernel.

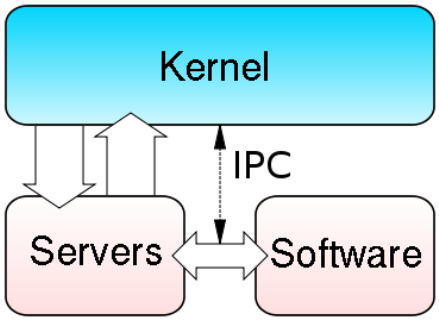
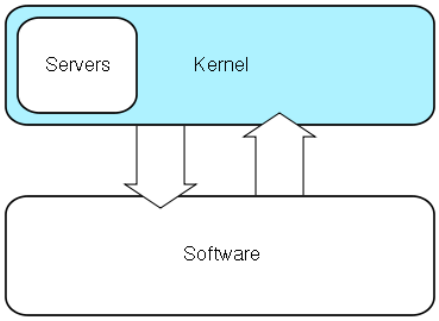


Illustrazione 14: Rappresentazione grafica di un Microkernel

3.6.4.Kernel Ibridi

I kernel ibridi sono essenzialmente dei microkernel che contengono all'interno dello spazio kernel codice non essenziale dal punto di vista del layer di collegamento all'hardware. Questo compromesso è finalizzato ad aumentare le prestazioni di alcune parti di codice che normalmente dovrebbero essere implementate nello spazio applicativo del sistema. Tale tecnica di progettazione fu adottata da molti sviluppatori di sistemi operativi prima che fosse dimostrato che i microkernel puri potevano avere invece performance più elevate. Tuttavia sono diversi i sistemi operativi moderni di questo tipo che impiegano di fatto microkernel modificati: Microsoft Windows è l'esempio più noto; ma anche XNU, il kernel di Mac OS X, è di fatto un microkernel modificato e anche DragonFly BSD adotta un'architettura a kernel ibrido. Spesso si sbaglia a considerare un kernel ibrido come un kernel monolitico in grado di caricare dei moduli, tale considerazione è errata poiché il comportamento di transito delle

informazioni tra spazio kernel e spazio applicativo nei kernel ibridi condivide concetti architetturali e meccanismi sia dei kernel monolitici che dei microkernel. La figura seguente semplifica la struttura del kernel ibrido.



3.6.5.Esokernel

Un esokernel si basa su un tipo di concetto completamente differente rispetto ai kernel finora introdotti. Un esokernel ha un'architettura che fornisce solamente la sicurezza nell'accesso alle risorse hardware, demandando la gestione allo sviluppatore a livello applicativo. Gli esokernel (o exokernel) risultano quindi di dimensioni molto contenute ed offrono allo sviluppatore una libertà ed un grado di gestione massimo. Un aspetto interessante di questi kernel è la possibilità di avere all'interno dello stesso sistema delle *libOS* diverse. Un sistema così progettato può, in linea di principio, eseguire programmi che facciano uso di librerie di sistema di Unix e di Windows. Attualmente gli esokernel sono utilizzati più in ambito di ricerca che su applicazioni commerciali. Di seguito la figura che ne schematizza il concetto.

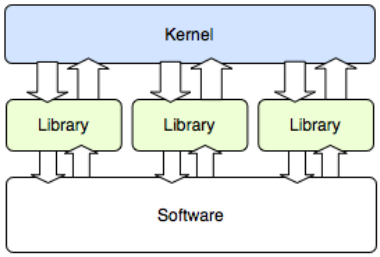


Illustrazione 16: Rappresentazione grafica di un exokernel

Inoltre esistono ancora sistemi senza uno specifico kernel cioè che delegano completamente il compito di interfacciarsi con l'hardware agli sviluppatori.

CAPITOLO 4

4.Linux Embedded

4.1.Introduzione

Nei capitoli precedenti è stato presentato il grande successo che hanno avuto, e stanno avendo tuttora, i sistemi operativi Linux-based nell'ambito dei sistemi embedded. Le motivazioni perché questo successo sia avvenuto e i pregi della scelta di un software open source sono stati largamente approfonditi. Si vuole ora entrare nel dettaglio delle caratteristiche di un sistema Linux per embedded, illustrare che cosa sia in realtà e cosa lo differenzia da uno standard Linux general purpose.

Per sistema Embedded Linux si intende un dispositivo basato su un microcontrollore di fascia alta solitamente dotato di MMU, anche se non strettamente necessario grazie a progetti come *μClinux*, capace di eseguire un sistema operativo basato su kernel Linux. La famiglia di processori a 32 bit più diffusa per questo scopo attualmente è quella con architettura ARM anche se esiste il supporto per moltissimi altri tipi di architetture. Ai vantaggi del software open source, già ampiamente discussi, si aggiungono i vantaggi tecnici propri di Linux come, ad esempio, il fatto di avere a disposizione un layer di gestione dell'hardware molto evoluto e consolidato.
Un importante considerazione di carattere tecnico che non è stata fatta in precedenza, va invece fatta per quelli che sono i limiti di un embedded Linux, o di un sistema operativo in generale: la presenza di un OS consente di delegare a questo molti dei compiti della gestione delle risorse hardware ma si traduce nella non completa padronanza delle tempistiche da parte del programmatore software. Il limite dunque è la difficoltà nell'ottenere un sistema hard Real-Time, termine che analizzeremo nello specifico più avanti nel capitolo, fornendo delle alternative che provano a risolvere questo problema. Quindi per tutti quei dispositivi dove si richiede una precisione elevata nelle tempistiche da rispettare è probabilmente più semplice affidarsi ad un microcontrollore privo di sistema operativo, gestendo direttamente le interrupt ed eventuali delay in modo preciso e controllato.
Riassumendo possiamo affermare che i dispositivi Embedded Linux riescono a ricoprire tutte le applicazioni più comuni, essendo molto flessibili, personalizzabili e versatili; per quanto riguarda invece applicazioni critiche hard real-time è preferibile l'impiego di un dispositivo senza OS o che impieghi un sistema operativo strettamente real-time (RTOS).

Un sistema operativo Linux embedded rispetto al classico Linux in versione desktop presenta caratteristiche simili nella struttura ma diverse dimensioni. Un sistema Linux tradizionale contiene all'interno moltissimi driver e componenti per garantire il supporto ad un numero enorme di periferiche hardware mentre un sistema Linux embedded è sempre reso essenziale mantenendo solo quei driver e quelle componenti necessarie ad una specifica configurazione hardware.

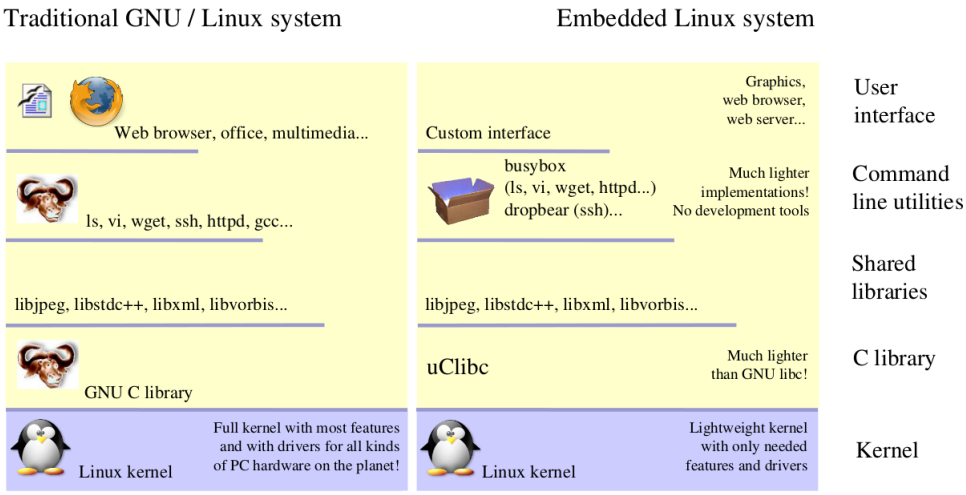


Illustrazione 17: Confronto tra Linux embedded e Linux tradizionale

4.2.Caratteristiche

Nelle prossime sezioni verranno presentate le principali caratteristiche di Linux embedded divise in sei aree di interesse. La scelta di sei ambiti è stata fatta per cercare di affrontare trasversalmente l'utilità di Linux embedded in tutti i sistemi. L'obiettivo è quello di trattare argomenti che emergono in ogni processo di sviluppo di un sistema embedded, senza tralasciare nemmeno considerazioni economiche che pur essendo non tecniche ricoprono notevole interesse [18].

4.2.1.Configurabilità

Essere in grado di personalizzare un sistema operativo per le esigenze particolari di un dispositivo è una caratteristica importante nei sistemi embedded. Solo per fare un esempio, non ha senso riempirsi di linee di codice per il supporto ad un interfaccia grafica se nemmeno c'è la necessità di averla. Ridurre all'osso un kernel fino alla dimensione più piccola possibile pur mantenendo il supporto che serve è una delle prime sfide che un designer si trova a dover affrontare.

Configurare Linux embedded non è molto differente da configurare un normale sistema Linux per desktop. Il processo di costruzione dell'OS si basa su un semplice file di testo per selezionare quali componenti includere. Dividendo il kernel in moduli componibili invece di mantenere una struttura monolitica consente di selezionare quali moduli caricare in fase di boot o perfino quali essere caricati dinamicamente durante il runtime, il che consente di utilizzare componenti opzionali solo quando è strettamente necessario [19]. Queste comodità di configurazione sono essenziali nei sistemi embedded, infatti oltre a poter caricare o scaricare dinamicamente moduli è possibile anche eseguire degli aggiornamenti parziali sul kernel. Uno svantaggio della struttura a moduli di Linux è l'instabilità potenziale che ne deriva. Quando i moduli sono dinamicamente integrati nel kernel hanno un accesso completo al sistema operativo senza però beneficiare della protezione della memoria questo però non costituisce più un problema in quei dispositivi sprovvisti di MMU.

4.2.2.Performance Real Time

Un sistema in tempo reale può essere definito come un apparato elaborativo in cui le varie attività computazionali devono essere realizzate entro predefiniti vincoli temporali e il cui grado di funzionalità non dipende solo dalla correttezza formale dei risultati delle elaborazioni ma anche dal tempo necessario a produrli. Il termine real inoltre richiama il fatto che il tempo di sistema deve essere adattabile alla scala temporale utilizzata per misurare i tempi tipici nell'ambiente in cui il sistema è impiegato; ad esempio se si deve gestire un fenomeno nella scala dei microsecondi, anche il sistema dovrà presentare caratteristiche di risoluzione e precisione almeno dello stesso ordine di grandezza se non superiori.

Se un sistema deve sottostare rigidamente ai vincoli temporali e una risposta oltre la deadline è interpretata come un fallimento si dice che il sistema è *hard real-time*, solitamente questo implica anche il fatto che le conseguenze alle risposte oltre i tempi stabiliti sono rischiose per l'ambiente in cui il sistema viene impiegato. Questi sistemi si impiegano in ambienti altamente critici. Quando invece si può transigere un certo tempo di risposta oltre la deadline, cioè se viene superato il limite imposto non si interpreta questo come un fallimento del sistema, si parla di *soft real-time*. Questi sistemi si impiegano in ambienti dove le risposte leggermente oltre i limiti non comportano conseguenze gravi. La stessa distinzione si può avere per i singoli processi che possono o meno avere esigenze di real-time.

Quando si affronta un progetto hardware/software in ambito industriale si deve cercare di soddisfare determinati requisiti di correttezza, efficienza, affidabilità, flessibilità, portabilità e riusabilità. Anche la predicibilità va tenuta in considerazione a seconda di quanto deve

essere stringente per il campo di applicazione interessato. Tutti questi aspetti interessano il sistema operativo nella sua totalità, ma un componente ricopre un ruolo fondamentale nella gestione complessiva dell'esecuzione dei vari processi e nell'organizzazione delle attività per soddisfare le specifiche: lo scheduler. Lo scheduler (che verrà presentato più precisamente nel paragrafo 5.3.1) si è evoluto con le varie versioni del kernel Linux e con la versione 2.6 si sono introdotte alcune migliorie che lo rendono molto più valido che in passato in applicazioni dove il tempo di risposta è importante.

Per il kernel Linux nella versione 2.6:

- sono stati inseriti dei punti di prelazione, nei quali cioè può girare lo scheduler; prima potevano verificarsi problemi in quanto un processo a priorità molto alta (e dalle specifiche temporali stringenti) poteva essere ritardato di molto in attesa che una *system call* di un altro processo terminasse.
- È stato ottimizzato lo *scheduler* in modo da ridurre l'*overhead*[9] introdotto dal sistema operativo, soprattutto in presenza di molti processi.
- Si possono avere anche sistemi privi di memoria virtuale, allargando di molto l'elenco dei processori supportati.
- È stata introdotta la possibilità di usare i *mutex*[10] senza ricorrere a chiamate di sistema.
- È stata introdotta nel kernel la possibilità di supportare i *timer* ad alta risoluzione POSIX ed è stato ottimizzato il meccanismo di gestione delle *thread*.

Nonostante tutti questi miglioramenti Linux rimane sempre un sistema che non è hard real-time. Per ottenere prestazioni real-time si può però ricorre a delle patch del kernel, delle quali una delle più utilizzate è RTAI, sviluppata dal Politecnico di Milano (www.rtai.org). Questa modifica prevede l'inserimento di un secondo kernel, di tipo hard real-time, che prende il controllo dell'hardware interponendosi tra il kernel originale e la macchina, facendo girare Linux come una sua applicazione a priorità più bassa. Così il controllo diretto degli interrupt lo ha solo RTAI e questo permette di aggirare le limitazioni di Linux ed avere un sistema veramente real-time.

9 Tempo necessario ad eseguire in CPU operazioni che non riguardano i processi
10 Semafori binari

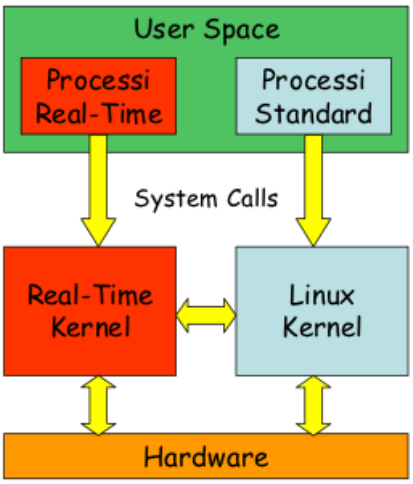


Illustrazione 18: Funzionameto di kernel linux con RTAI

4.2.3.Graphical User Interface

Così come i normali PC si basano su GUI invece che su interfacce testuali anche molti sistemi embedded le stanno adottando. Un'interfaccia grafica per sistemi embedded può assumere diverse forme, come piccoli schermi LCD o un uscita VGA o perfino un schermo orientabile ad alta risoluzione. Anche se il concetto è lo stesso dei PC, quando parliamo di sistemi embedded vanno tenuti in considerazione alcuni requisiti che sono unicamente loro. Linux ha sempre sofferto di una frammentazione per quanto riguarda le GUI e nonostante X-Windows sia la cosa che si avvicina di più ad uno standard, ci sono molte altre alternative popolari. Lineo Systems è un fornitore di alcune versioni embedded di Linux con supporto commerciale e una delle opzioni ce forniscono è quella di un kit su misura per costruire una GUI per sistemi embedded [20]. Microwindows and NanoGUI sono due progetti open source che sono stati accorpati per fornire una libreria gratuita per emulare le API di Win32 in ambiente Linux [21]. Esiste anche un'altra strada che molti sviluppatori di sistemi embedded hanno scelto di percorrere quando si parla di GUI. Molti sistemi includono un web server che offre agli utenti un'interfaccia attraverso HTML. Questo consente dall'esterno di accedere al dispositivo da qualsiasi luogo attraverso un web browser e svincola totalmente il dispositivo dal possedere un display.

4.2.4.Strumenti di Sviluppo

Gli sviluppatori di Linux embedded possono avvantaggiarsi di una serie di strumenti di sviluppo che provengono dal mondo PC. Infatti la collezione dei compilatori, debugger,

assembler e linker di GNU hanno una vastissima applicazione per i microprocessori negli embedded. La possibilità di avere degli strumenti così affidabili è uno dei vantaggi principali derivati dalla comunità open source. Non solo sono disponibili gratuitamente, testati e migliorati da molti anni ma consentono anche di avere la libertà di adattarli ad un nuovo modello di microprocessore se questo non è ancora stato implementato.

4.2.5.Considerazioni Economiche

Uno dei principali fattori di attrazione di Linux è che è completamente gratuito. Sebbene si possano sempre comprare versioni commerciali da molte società per avere customizzazioni e funzioni aggiuntive il codice sorgente di Linux si può scaricare gratuitamente grazie anche all'applicazione della licenza GPL. Nel campo dei sistemi embedded è frequente trovare distributori che pretendono delle royalties sulla vendita dei dispositivi ma la strada del software libero è sempre percorribile. Vanno tenuti in considerazione però molti aspetti quando si vuole prendere una decisione di tipo “build or buy” ad esempio va considerato il tempo di messa sul mercato che cambia notevolmente nel caso si debba sviluppare in casa il software necessario o se viene acquistato da fuori.

4.2.6.Supporto Tecnico

Uno dei preconcetti più diffusi che le persone hanno verso Linux e i sistemi operativi che non vengono distribuiti da un unica azienda è il fatto che il supporto tecnico sia assente. In realtà ci sono possibilità molto più alte di veder risolto il proprio problema quando ci si riferisce ad una comunità open source che quando si cerca di percorrere i canali messi a disposizione delle aziende di software proprietario. Questo vantaggio è insito nella struttura del software open source e, come già citato in precedenza, è più facile trovare una soluzione ad un problema in una comunità dove ci si può confrontare con chi può aver avuto problemi simili che attendere l'interesse di una azienda che mantiene solo il software che crea degli utili.

4.3.Esempi Applicativi

In rete si nota molto fermento attorno all'impiego di sistemi Linux embedded in moltissimi campi. L'espansione è dovuta forse anche al boom dell'open hardware di questo ultimo periodo (Arduino, RaspberryPI, System On a Chip (SoC) a basso costo, ecc.) che si accompagna alla “makers revolution” di cui tanto si parla. Ciò che inizialmente era abbastanza limitato al settore dell'hobbystica e dell'elettronica fai da te si sta trasferendo nel mondo industriale. Nell’industria infatti i sistemi operativi Linux embedded trovano numerose

applicazioni: una tra le tante è la videosorveglianza, una tecnologia sempre più adottata, anche nel nostro Paese, per ragioni di sicurezza, sia nei reparti di produzione delle aziende che nei punti vendita. Tale tecnologia oggi va oltre i tradizionali sistemi di telecamere a circuito chiuso e tende a orientarsi verso le più versatili soluzioni basate su protocollo Ip. Un esempio di telecamere sono le 'network camera', fornite da Axis Communications e utilizzate in molte realizzazioni anche nell'area retail, all'interno di supermercati e ipermercati. Tali telecamere non hanno bisogno della connessione a un pc, poiché integrano già un Web server, gestito da una versione di Linux embedded, e dispongono di una porta Ethernet per il diretto collegamento in rete. Sono controllabili da qualunque postazione remota e in grado di notificare gli allarmi e inviare immagini al responsabile del controllo via e-mail o tramite i protocolli Tcp, Http e Ftp [22].

In un settore molto diverso, quale quello dell'automotive, voci di corridoio rivelano che la Toyota Lexus IS in uscita nel 2014 è la seconda automobile ad offrire un sistema di infotainment basato interamente su Linux. Mentre le ricerche di ABI prevedono che Linux crescerà velocemente oltre il 20% nei computer di bordo del settore automotive avvicinando il primato di Microsoft nel 2018 [23]. Anche nel futuristico campo delle automobili autonome Linux sta trainando il settore [24]. L'auto autonoma di Google monta Linux e anche prototipi di General Motors e Volkswagen. Questo mercato diventerà vastissimo, Navigant Research prevede che le vendite di veicoli autonomi sorpasseranno i 95 milioni di unità nel 2035. La Google car basata su un sistema Ubuntu-like ha già percorso 500mila miglia senza un pilota umano senza causare incidenti per le strade della California.

CAPITOLO 5

5.Android

5.1.Storia

Android Inc. è stata fondata nell'ottobre del 2003 da Andy Rubin, Rich Miner, Nick Sears e Chris White. Inizialmente la società operò in segreto, rivelando solo di progettare software per dispositivi mobili. La svolta nello sviluppo di Android arriva nel luglio del 2005 quando Google acquista Android Inc., trasformandola nella *Google Mobile Division* mettendo a capo sempre Andy Rubin. Da questa data in poi il team comincia a sviluppare un sistema operativo per dispositivi mobili basato sul kernel Linux. La presentazione ufficiale venne fatta il 5 novembre del 2007 dalla nuova OHA (*Open Handset Alliance*) un unione di operatori telefonici, produttori di dispositivi mobili, produttori di semiconduttori, compagnie di sviluppo software e di commercializzazione, avente come scopo quello di creare standard aperti per dispositivi mobili. L'OHA include Google, produttori di smartphone come *HTC* e *Samsung*, operatori di telefonia come *Sprint Nextel* e *T-Mobile* e produttori di processori come *Qualcomm* e *Texas Intruments*. Insieme al lancio ufficiale del sistema operativo venne rilasciato anche il primo *Software Development Kit* (SDK) per gli sviluppatori (il quale includeva gli strumenti di sviluppo, le librerie, un emulatore del dispositivo, la documentazione e alcuni progetti di esempio).

Il primo dispositivo equipaggiato con Android lanciato sul mercato fu L'*HTC Dream*, il 22 ottobre del 2008. Dal 2008 in poi il successo di questo sistema operativo ne ha garantito un rapido sviluppo e il continuo rilascio d nuove funzionalità. La versione più recente rilasciata dalla casa di Mountain View è la 4.4 KitKat[11].

5.2.La struttura del sistema

La struttura del sistema è sostanzilmente uno stack [25] di software Open Source, rilasciato sotto licenza *Apache*; gli sviluppatori hanno accesso all'intero codice sorgente, consentendo una rapidissima portabilità da un dispositivo ad un altro. Nello stack sono inclusi il sistema operativo, applicazioni middleware (intermediarie fra app e componenti software) e applicazioni chiave (fondamentali per il corretto funzionamento delle funzionalità di base del sistema offerte all'utente finale).

Android è un sistema operativo progettato per adattarsi ai dispositivi mobili, molta attenzione

11 Le versioni del sistema sono indicate a livello ufficiale da un numero di versione seguito sempre da un nome in codice per tradizione ispirato a prodotti dolciari sempre in ordine alfabetico.

è stata dedicata ad alcuni aspetti fondamentali in questo campo; ad esempio la riduzione dei consumi energetici, per fornire una maggiore autonomia ai dispositivi; le ridotte dimensioni in termine di spazio occupato in memoria, per poter garantire prestazioni fluide anche con hardware ridotto; la portabilità, in modo da poter essere impiegato in ogni dispositivo e adattato a questo.

Fra le features troviamo:

- Application framework, che consente il riutilizzo e la sostituzione dei componenti;
- *Dalvik virtual machine*, macchina virtuale ottimizzata per dispositivi mobili;
- Browser integrato basato sul motore open source WebKit;
- Grafica ottimizzata, supportata da un'apposita libreria grafica per il 2D, la grafica 3D è basata sulla specifica OpenGL ES, recentemente aggiornati alla versione 3.0 con Android 4.3;
- SQLite, motore per la memorizzazione di strutture di dati;
- Supporto multimediale per i più comuni formati audio, video e di immagine;
- Telefonia GSM (legata all'hardware);
- Bluetooth, EDGE, 3G, LTE, NFC e WiFi (legati all'hardware);
- Fotocamera, GPS, bussola e accelerometro (legati all'hardware);
- Ricco ambiente di sviluppo che include un emulatore, strumenti per il debug, profili adatti alla gestione della memoria e delle prestazioni e un plugin per l'IDE Eclipse.

Nel seguente schema vengono mostrati i componenti principali del sistema Android.

Come evidenziato in figura lo stack software che compone Android è l'insieme di un kernel Linux, una serie di librerie in linguaggio C/C++ rese accessibili attraverso un framework sviluppato in Java e di un insieme di applicazioni.
Il kernel Linux impiegato attualmente è quello della versione 3.4 (anche se si è partiti dal 2.6), usato per i servizi di base del sistema, quali sicurezza, gestione della memoria, gestione dei processi, di rete, e dei driver. Esso funziona anche come un livello di astrazione tra l'hardware e il resto dello stack software.
La parte che distingue Android dall'essere un semplice adattamento di un sistema operativo Linux è l'ambiente di *runtime*. Esso costituisce il motore che permette l'esecuzione delle applicazioni e, assieme alle *core libraries*, costituisce la base del livello software superiore. Nello specifico, ogni applicazione che viene lanciata crea un proprio processo con associata un'istanza della *Dalvik virtual machine* (macchina virtuale creata appositamente per la gestione efficiente di più applicazioni contemporanee). Quest'ultima fa riferimento al kernel sottostante per funzionalità di basso livello come gestione di *thread* e della memoria.
L'*application framework* invece mette a disposizione le classi utilizzate per creare le applicazioni, fornendo contemporaneamente un interfaccia per l'accesso all'hardware, la gestione dell'interfaccia utente e delle risorse necessarie ai processi. Queste funzionalità sono fornite da un insieme di servizi, i quali includono un insieme di *View* per costruire le applicazioni, *Content Providers* (per comunicare fra applicazioni), *Resource Manager* (per l'accesso alle risorse grafiche), *Notification Manager* (per la gestione dei messaggi e delle notifiche a video) e *Activity Manager* (per la gestione dei cicli di vita delle applicazioni). L'architettura è così strutturata per poter semplificare il riutilizzo di ogni singolo componente (ricondivisibile e utilizzabile fra applicazioni diverse).

5.3.Il Kernel di Android

Come visto precedentemente, Android ha come fondamenta la versione 3.4 del kernel Linux. Il kernel è la parte centrale, il nucleo, la più importante dell'intero sistema. Per la descrizione del kernel in generale e delle sue tipologie si rimanda al paragrafo 3.6.
Il kernel Linux viene detto modulare, infatti pur avendo una struttura monolitica, supporta la capacità di caricare o rilasciare in modo dinamico porzioni di codice dette moduli che consentono l'integrazione di nuove funzionalità non appena queste vengono richieste. Esso quindi racchiude i vantaggi di un kernel monolitico garantendo al contempo una migliore occupazione di memoria, al costo di un irrisorio decremento prestazionale dovuto al tempo di caricamento dei moduli.

Quando si parla di kernel è necessario precisarne le differenze con le applicazioni comuni. Una prima distinzione si evidenzia in quelli che vengono definiti i *contesti di esecuzione*. I sistemi moderni sono dotati dell'unità di protezione della memoria, la Mamory Management Unit (MMU), e permettono di rendere disponibile al kernel un ambiente indicato come *kernel-space*, composto da uno spazio di memoria privato ed un completo accesso alle funzioni hardware. Le applicazioni invece sono eseguite in un ambiente detto *user-space*, con delle limitazioni tali da costringerle ad intervenire sul sistema solo tramite l'intervento del kernel, tramite le già citate *system calls*. Questo comportamento salvaguarda la protezione di zone delicate della memoria e favorisce una maggiore stabilità del sistema.
Ci sono altre distinzioni dalle applicazioni comuni come ad esempio: il kernel non contiene le librerie C e gli header standard ma ne possiede una versione ridotta per una questione di memoria; é implementato in linguaggio GNU C e non propriamente in C; non è in grado di eseguire operazioni in virgola mobile; ha uno stack di dimensione piccola e finita per ciascun processo che non è incrementabile come per le applicazioni.
Gli aspetti implementativi del kernel Linux sono molto complessi e non è dunque semplice conoscere in dettaglio le oltre 6 milioni di righe di codice che lo compongono, tuttavia possiamo individuare cinque blocchi funzionali che lo costituiscono:

- il process scheduler, che controlla e permette l'accesso dei processi all'unità centrale di elaborazione;
- il memory manager, che gestisce l'uso della momoria dei processi fornendo un accesso sicuro e controllato;
- il virtual filesystem, che ha il compito di riassumere alcuni dettagli dei dispositivi hardware attraverso dei file che ne rappresentano un interfaccia;
- la network interface, che fornisce l'accesso ai protocolli di rete e al relativo hardware;
- l'inter-process-communication, che ha il compito di gestire i meccanismi di comunicazione del sistema.

Infine il kernel fornisce un ampio insieme di moduli che conoscono come gestire operazioni di basso livello legate strettamente all’hardware.

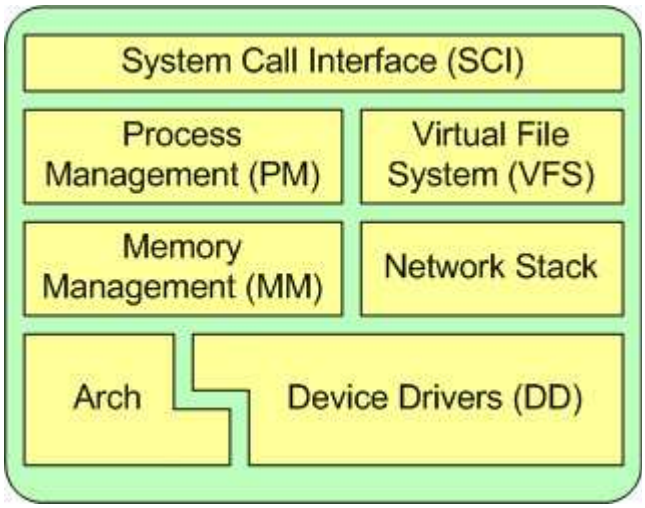


Illustrazione 20: Struttura del Kernel Linux

5.3.1.Process Scheduler

Il process scheduler è un programma che in base ad una serie di richieste di accesso ad una risorsa, stabilisce l'ordine temporale ottimale con cui eseguire tali richieste [26]. La scelta avviene secondo alcuni parametri che tendono ad ottimizzare l'accesso alla risorsa da parte di un processo. L'attenzione ad alcuni parametri al posto di altri differenzia le varie *politiche di scheduling,* che si attuano tramite diversi algoritmi.

I parametri presi in considerazione sono:

- Cpu Utilization: tempo effettivo di utilizzo della CPU per l'esecuzione dei processi;
- Throughput: il numero di processi completati nell'unità di tempo;
- Turnaround time: il tempo che intercorre tra la sottomissione di un processo ed il completamento della sua esecuzione;
- Waiting time: il tempo in cui un processo pronto per l'esecuzione rimane in attesa della CPU;
- Response time: il tempo che trascorre tra la sottomissione del processo e l'ottenimento della prima risposta.

Alcuni degli algoritmi per lo scheduling sono:

- **FCFS**: First Come First Served, che esegue i processi nello stesso ordine in cui essi vengono sottomessi al sistema. Il primo processo ad essere eseguito è quello che chiede per primo la CPU e gli altri vengono messi in coda.
- **RR**: Round Robin, che esegue i processi nell'ordine d'arrivo, come il FCFS, ma può

eseguire la prelazione[12], ovvero quando un processo in esecuzione supera il *quanto di tempo* stabilito lo rimette in coda.

- **SJF**: Shortest Job First, una famiglia di algoritmi che si basano sul tempo stimato di utilizzo della CPU. Il sistema operativo utilizza i dati delle precedenti elaborazioni per stabilirlo secondo una formula[13].
- **Priority-Based**: gli algoritmi basati sulla priorità di ogni singolo processo.

In Linux lo scheduler si è evoluto con le versioni arrivando ad avere delle caratteristiche particolari che cercherò di sintetizzare nelle righe seguenti. Questo scheduler si chiama CFS (Completely Fair Scheduler) e l'idea principale dietro ad esso è di mantenere equilibrato il tempo di occupazione della CPU concesso ai processi. Quando non vi è questo bilanciamento, si sceglie di concedere ai processi non bilanciati del tempo per essere eseguiti. Per determinare il bilanciamento, lo scheduler mantiene memorizzato il tempo dato ad un processo in quello che viene chiamato *virtual runtime*. Più è piccola questa struttura (e quindi meno tempo è stato concesso al processo), più avrà priorità per il controllo del processore. Viene implementato inoltre il concetto di equità anche per i processi in attesa (per esempio per quelli in attesa di I/O), concedendogli una giusta porzione di tempo di esecuzione quando ne hanno bisogno. A differenza di ciò che accadeva con i precedenti tipi di scheduler (i processi erano mantenuti in code d'esecuzione), il CFS mantiene i processi in un albero rosso-nero con ordinamento temporale. La struttura rosso-nera porta con sé importanti proprietà: è auto-bilanciata (non ci sono percorsi nell'albero più lunghi di due elementi rispetto agli altri) e ogni operazione sull'albero ha un costo di O(log n) (con n nodi dell'albero), quindi inserimenti e cancellazioni sono molto rapidi e efficienti.

12 Lo scheduler può sottrarre il possesso della CPU al processo in esecuzione anche se questo potrebbe proseguire. Si parla di scheduling *preemptive, non-preemtive* nel caso in cui non possa intervenire.

13 $Tn+1=\alpha*tn+(1-\alpha)\ Tn$ (dove $0<=\alpha<=1$)

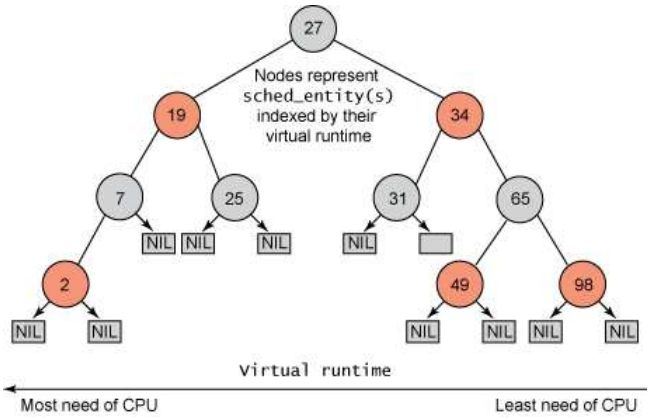


Illustrazione 21: Struttura dati utilizzata nello scheduling CFS

Processi con maggior bisogno di utilizzare la CPU (minor virtual runtime) sono mantenuti nella parte sinistra dell'albero, viceversa processi con meno bisogno (maggior virtual runtime) sono mantenuti nella parte destra. Lo scheduler, seguendo il principio di equità, prende di volta in volta il processo più a sinistra. Concluso l'istante di tempo concessogli, viene sommato tale tempo al virtual runtime del processo (che verrà reinserito nell'albero se non si è ancora completato). Estraendo un nodo, l'albero, per auto-bilanciarsi, tenderà a spostare via via i nodi più a destra verso sinistra, così da mantenere equità nelle successive scelte dei processi che dovranno essere eseguiti. A differenza di altre versioni di scheduler, il CFS non utilizza il concetto di priorità direttamente, bensì un fattore di "decadimento" legato al tempo che un processo può rimanere in esecuzione. Processi a bassa priorità hanno un più alto valore di tale fattore, viceversa per quelli ad alta priorità. Questo significa che il tempo che un processo può rimanere in esecuzione diminuisce più velocemente per un processo a bassa priorità piuttosto che per uno ad alta priorità. Un'ulteriore novità introdotta con il CFS è il concetto di gruppo di scheduling. Anche in questo caso l'obiettivo è l'equità ed è particolarmente indicato nei casi di processi che ne avviano molti altri. In questi casi invece di trattare separatamente ogni processo li si prende in gruppo. Il processo che li ha lanciati condivide con il gruppo (gestito in una gerarchia) il proprio virtual runtime, mentre ogni singolo processo continua a mantenere il proprio virtual runtime in modo indipendente [27].

5.3.2.Memory Manager

Il gestore della memoria si preoccupa di allocare, deallocare e gestire la memoria che viene assegnata ai processi, gestisce inoltre la memoria virtuale, ovvero quella che viene trasferita dalla RAM al disco fisso [28]. Deve garantire la protezione di quelle parti della memoria dove

sono presenti i moduli del sistema e impedire ai processi di accedervi, oppure tenere separate le parti di memoria dei vari programmi che sono contemporaneamente in esecuzione. L'allocazione della memoria deve essere un'operazione invisibile al processo che non deve dipendere in alcun modo dalla posizione in cui sono allocati i suoi dati, ma allo stesso tempo il processo deve considerare come contigue tutte le sezioni di memoria che gli appartengono (codice e dati). Inoltre in un sistema multiprogrammato, che prevede l'esecuzione di più istanze dello stesso programma, deve garantire la condivisione del codice ma mantenere separati i dati.

5.3.3.Virtual Filesystem

Il virtual filesystem è un componente software che fornisce un livello di astrazione dei file concreti in modo da garantirne un acceso che sia indipendente dal supporto di memorizzazione. In Linux si applica il concetto di “Everything is a File” tramite un filesystem complesso che traduce virtualmente anche i supporti removibili, le periferiche e i vari dispositivi collegati. Tutto diventa un file e la gerarchia è un insieme di cartelle e sottocartelle. Android ha una struttura meno rigida e dipende dal dispositivo considerato ma ci sono una serie di cartelle fondamentali che sono sempre presenti, che sono rappresentate in figura 22.

Illustrazione 22: Struttura logica di base del filesystem di Android

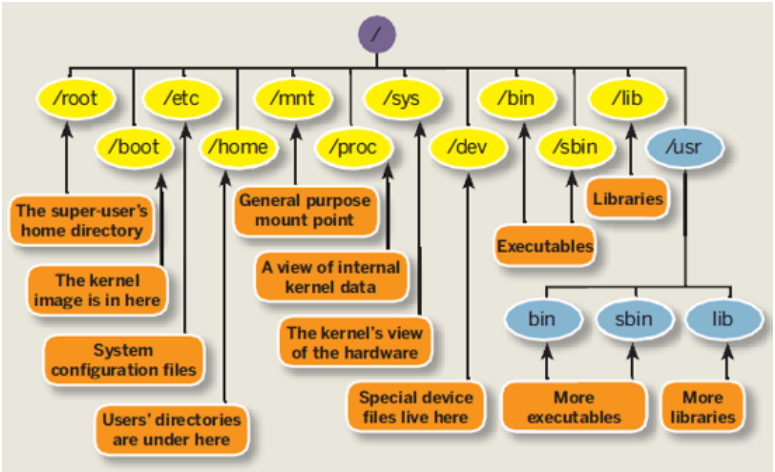


Illustrazione 23: FileSystem in Linux

5.3.4.Network Interface

L'interfaccia di rete fornisce l'accesso ai protocolli di rete e al relativo hardware.

5.3.5.Inter Process Communication (IPT)

Con l'espressione *comunicazione tra processi* ci si riferisce a quel software in grado di consentire lo scambio di dati e/o informazioni, tutti i sistemi operativi fornisco alcuni strumenti perché questo sia possibile. I concetti fondamentali messi a disposizione nell'IPC sono i segnali, le pipe, lo scambio di messaggi, i semafori, le chiamate a procedure remote e i socket.

5.4.Diffusione

Per testimoniare la rapidissima ascesa di Android tra i primi sistemi operativi per sistemi embedded a livello mondiale è utile riportare qualche dato. Considerando la difficoltà con cui si può tenere traccia del fenomeno che ha un espansione velocissima ho giudicato interessante riportare i dati relativi al mercato degli smartphones, settore a cui viene dedicata particolare attenzione da parte dei media dato che fa parte del mondo consumer e si avvicina agli interessi della gente, e di cui è possibile trovare analisi di mercato approfondite.
Secondo una analisi di Strategy Analytics, Android ha rappresentato il 70% di tutti gli smartphones spediti globalmente nel 2012 raggiungendo la quota di quasi mezzo miliardo di dispositivi venduti solo durante quell'anno [29]. Più aggiornati i dati dell'analisi di International Data Corporation (IDC) Worldwide Quarterly Mobile Phone Tracker [30] che per il secondo quadrimestre del 2013 hanno contato circa 190 milioni di unità vendute per Android che raggiunge circa l'80% del valore di tutto il mercato in quel periodo. Nella tabella seguente tutti i dati.

Operating Systems	2Q13 Unit Shipments	2Q13 Market Share	2Q12 Unit Shipments	2Q12 Market Share	Year over Year Change
Android	187.4	79.3%	108	69.1%	73.5%
iOS	31.2	13.2%	26	16.6%	20%
Windows Phone	8.7	3.7%	4.9	3.1%	77.6%
BlackBarry OS	6.8	2.9%	7.7	4.9%	-11.7%
Linux	1.8	0.8%	2.8	1.8%	-35.7%
Symbian	0.5	0.2%	6.5	4.2%	-92.3%
Others	N/A	0.0%	0.3	0.2%	-100%
TOTAL	**236.4**	**100%**	**156.2**	**100%**	**51.3%**

Tabella 2: Top Smartphone Operating Systems, Shipments, and Market Share, Q2 2013 (Units in Millions)

CONCLUSIONI

Questa tesi si è posta come obiettivo quello di presentare una panoramica sul software Open Source impiegato nei sistemi embedded, illustrando le principali alternative ed i vantaggi di una scelta orientata in questo senso. Cercare di far luce in un campo così vasto come quello dei sistemi embedded non si è rivelato facile ed, ancor di più, cercare di trovare dei dati aggiornati sull'andamento del software libero in un settore in grande sviluppo. Si avverte un forte fermento per le soluzioni che impiegano Linux Embedded sempre più diffusamente e le comunità che girano attorno al mondo Linux crescono di dimensione e di credibilità. Si è cercato di fornire una visione d'insieme del fenomeno concentrandosi su dati effettivi e presentando le caratteristiche di base di Linux Embedded e del fratello Android. Come testimoniano i dati, la crescita di strumenti Open Source applicati in campo embedded continuerà ad aumentare conquistando sempre nuovi ambiti di applicazione, dovuto ai molti vantaggi che questo approccio porta con se.

BIBLIOGRAFIA

[2] C.Ebert e C.Jones, "Embedded software: Facts, figures, and future", Computer 42, no. 4 (2009).

[4] A. Malinowski, H. Yu, "Comparison of Embedded System Design for Industrial Application", IEEE Transaction on Industrial Informatics, Vol. 7, N. 2, Maggio 2011

[5] J. J. Rodriguez-Andina, M. J. Moure, and M. D. Valdes, “Features, design tools, and application domains of FPGAs”, IEEE Transaction on Industrial Electronics, vol. 54, no. 4, Agosto 2007.

[6] S. Williams, "Free as in Freedom. Richard Stallman's crusade for free software", O'Reilly Media, 2002

[7] Gruppo Ippolita, "Open non è Free, Comunità digitali tra etica hacker e mercato globale", Elèuthera, 2005

[8] L. Torvalds e D. Diamond, "Rivoluzionario per caso. Come ho creato Linux (solo per divertirmi)", Garzanti, 2001.

[9] E. S. Raymond, "The Cathedral and the Bazaar", O'Reilly Media, 1997

[19] J. Epplin, "Inside real-time Linux", Dr. Dobb's Journal, Marzo 2000

[22] G. Fusari, "Linux ‘si fa largo’ nel mondo embedded", Embedded 7, Luglio 2004

[27] M. Masiero, Android 4.0: overview, novità e funzionamento del sistesma operativo targato Google, 2012, Tesi di Laurea.

SITOGRAFIA

[1] Pagina di wikipedia relativa ai sistemi embedded : http://it.wikipedia.org/wiki/Sistema_embedded (visitato il 20/10/13)

[3] M. Cesati, Slide del corso Sistemi Embedded e Real-Time dal sito http://sert13.sprg.uniroma2.it/calendario.html (visitato il 19/10/13)

[10] Articolo BBC : http://news.bbc.co.uk/2/hi/programmes/click_online/8133068.stm (visitato il 20/10/12)

[11] Sito sulla Forrester Research : http://www.wuerth-phoenix.com/de/news/artikel/forrester-research-cambio-culturale-per-lopen-source-anche-in-italia/ (visitato il 22/10/12)

[12] Sito del Centro nazionale per l'informatica nella pubblica amministrazione : http://www.osspa.cnipa.it/home/ (visitato il 23/10/12)

[13] K. Yrvin e B. Stubert, "Embedded system design with open source software: doing it right", 19 Aprile 2010, http://www.embedded.com/print/4008931 (visitato il 2/11/13)

[14] Pagina ufficiale del compilatore llvm : http://llvm.org/Users.html (visitato il 28/10/13)

[15] Free Research from UBM Tech scaricabile da http://e.ubmelectronics.com/2013EmbeddedStudy/index.html (visitato il 30/10/13)

[16] Choosing Free Software in embedded systems, http://freeelectrons.com, 15 Settembre 2009

[17] Pagina di wikipedia realtiva al kernel : http://it.wikipedia.org/wiki/Kernel (visitato il 30/10/13)

[18] A. Tucker, "An Overview of Embedded Linux", Marzo 2000, http://www.cs.washington.edu/education/courses/cse585/00wi/project/emlinux.pdf

[20] Pagina del progetto Lineo : http://www.lineo.co.jp/modules/english/ (visitato il 12/11/13)

[21] Pagina di Nano-X Window System : http://www.microwindows.org/ (visitato il 6/11/13)

[23] E. Brown, "The rise of Linux in in-vehicle infotainment (IVI)", 24 Luglio 2013, http://linuxgizmos.com/linux-based-in-vehicle-infotainment-on-the-rise/ (visitato il 5/11/13)

[24] E. Brow, "Linux Leads Self-Driving Car Movement", 9 Settembre 2013, http://www.linux.com/news/embedded-mobile/mobile-linux/737295-linux-leads-self-driving-car-movement (visitato il 3/11/13)

[25] A. Al Salool, "Android software Stack & native application architecture", http://android-app-tutorial.blogspot.it/2012/08/architecture-system-application-satck.html (visitato il 2/11/13)

[26] Pagina di Wikipedia relativa allo scheduler : http://it.wikipedia.org/wiki/Scheduler (visitato il 6/11/13)

[28] Pagina di wikipedia sul gestore della memoria : http://it.wikipedia.org/wiki/Gestore_della_memoria (visitato il 6/11/13)

[29] R. Lehrbaum, "Half-a-billion Android smartphones shipped in 2012", 9 Marzo 2013, http://linuxgizmos.com/nearly-half-a-billion-android-smartphones-shipped-in-2012/ (visitato il 6/11/13)

[30] Pagina Ufficiale di IDC relativa ai sondaggi quadrimestrali sul mercato degli smartphones : http://www.idc.com/getdoc.jsp?containerId=prUS24257413 (visitato il 6/11/13)

ELENCO ILLUSTRAZIONI