TITOLO DEL DOCUMENTO

December 11, 2025

INDICE

1 Introduzione all'Ecosistema dei Pacchetti Arch Linux **2**
- 1.1 Obiettivo della Guida . . . 2
- 1.2 Struttura del Documento . . . 2

2 Repository Ufficiali e Pacman **3**
- 2.1 Come Funzionano i Repository Ufficiali . . . 3
- 2.2 Installazione dei Pacchetti Precompilati . . . 3
- 2.3 Vantaggi e Limitazioni . . . 4

3 AUR (Arch User Repository) **5**
- 3.1 Cos'è l'AUR . . . 5
- 3.2 Differenze tra AUR e Repository Ufficiali . . . 5
- 3.3 Ruolo degli Helper AUR (yay, paru, ecc.) . . . 6

4 Il PKGBUILD **6**
- 4.1 Struttura Generale del File . . . 6
- 4.2 Variabili Principali . . . 7
- 4.3 Funzioni build() e package() . . . 8
- 4.4 Gestione delle Dipendenze . . . 8
- 4.5 Generazione della Versione tramite pkgver() . . . 9

5 Pacchetti -git **9**
- 5.1 Caratteristiche dei Pacchetti Basati su Repository Git . . . 9
- 5.2 Versionamento Automatico . . . 10
- 5.3 Differenze con i Pacchetti Stabili . . . 10

6 Conflitti tra Pacchetti **11**
- 6.1 Come Nascono i File Conflicts . . . 11
- 6.2 Perché hyprland e hyprland-git Confliggono . . . 11
- 6.3 Strategie di Risoluzione . . . 11

7 Conclusioni **12**
- 7.1 Riepilogo dei Concetti Chiave . . . 12
- 7.2 Passi Successivi . . . 13

1 Introduzione all'Ecosistema dei Pacchetti Arch Linux

1.1 Obiettivo della Guida

Questa guida ha l'obiettivo di fornire una comprensione completa e dettagliata del funzionamento dell'ecosistema dei pacchetti in Arch Linux, coprendo sia i pacchetti dei repository ufficiali gestiti tramite `pacman`, sia i pacchetti dell'AUR installati tramite helper come `yay`.

In particolare, la guida permette di comprendere:

- Il flusso operativo completo di `pacman`: come vengono scaricati i pacchetti dai repository ufficiali, come vengono verificate firme e checksum, come vengono risolte le dipendenze e come vengono installati i file nel filesystem.
- Il ruolo dell'AUR: cosa contiene, come si differenzia dai repository ufficiali e quali sono le responsabilità dell'utente durante il build dei pacchetti.
- La struttura e il funzionamento di un `PKGBUILD`: variabili principali, funzioni (`prepare()`, `build()`, `check()`, `package()`, `pkgver()`) e come queste influenzano la creazione del pacchetto.
- Il processo completo di `makepkg`: gestione delle sorgenti, controllo dei checksum, build dei binari, popolamento della directory `$pkgdir` e generazione del pacchetto finale.
- Le differenze tra pacchetti stabili e pacchetti `-git`, come viene calcolata la versione dinamica e come gli helper AUR gestiscono automaticamente le dipendenze e la costruzione.
- La gestione dei conflitti tra pacchetti, inclusi file condivisi, pacchetti che forniscono lo stesso binario o libreria, e strategie per risolverli.
- Principi di sicurezza pratici: verifica dei PKGBUILD, controllo delle sorgenti, chiavi PGP, uso di checksum, e rischi legati a eseguire script sconosciuti durante il build.

Questa sezione serve quindi a creare una base solida, permettendo al lettore non solo di capire il c̈omeë dei comandi `pacman` e `yay`, ma soprattutto il p̈erchéë di ogni fase e decisione all'interno del processo di installazione e build dei pacchetti Arch Linux.

1.2 Struttura del Documento

Il documento
‘e organizzato per guidare il lettore dal livello introduttivo fino alla comprensione avanzata dei pacchetti Arch Linux e del loro ciclo di vita, con un focus sia sui repository ufficiali sia sull'AUR. Ogni sezione
‘e strutturata in modo da fornire prima concetti teorici e contesto, seguiti da dettagli tecnici operativi, esempi pratici e best practice.

La struttura generale
‘e la seguente:

- **Introduzione:** presenta gli obiettivi della guida, i termini chiave, e il contesto in cui si inserisce l'ecosistema dei pacchetti Arch Linux.
- **Repository ufficiali e pacman:** descrive come pacman interagisce con i repository ufficiali, il flusso di installazione dei pacchetti precompilati, la risoluzione delle dipendenze, il controllo delle firme e la gestione dei conflitti.
- **AUR (Arch User Repository):** spiega cosa
‘e l'AUR, come differisce dai repository ufficiali, quali file contiene, e il ruolo degli helper AUR come yay e paru nella gestione del build e dell'installazione dei pacchetti.
- **Il PKGBUILD:** analizza in dettaglio le variabili principali, le funzioni chiave (`prepare()`, `build()`, `check()`, `package()`, `pkgver()`) e come queste influenzano la creazione del pacchetto.
- **makepkg e processo di build:** illustra l'intero ciclo di costruzione di un pacchetto, dalla fetch delle sorgenti alla generazione del file .pkg.tar.zst finale, inclusi checksum, firma, e gestione delle dipendenze di build.

- **Pacchetti -git e versionamento dinamico:** spiega la differenza tra pacchetti stabili e -git, come viene calcolata dinamicamente la versione tramite `git describe`, e come gli helper AUR gestiscono le dipendenze e la build automatica.
- **Conflitti tra pacchetti:** approfondisce i casi in cui due pacchetti possono avere file in comune, come pacman gestisce i conflitti, e le strategie per risolverli.
- **Sicurezza e best practice:** linee guida per analizzare un PKGBUILD prima della build, verifica dei sorgenti, uso dei checksum, chiavi PGP e rischi di eseguire script esterni.
- **Esempi pratici e appendici:** include esempi commentati di PKGBUILD, sia per pacchetti stabili sia -git, e una checklist di comandi utili per il monitoraggio, il debug e la manutenzione dei pacchetti.

Questa struttura permette al lettore di seguire un percorso progressivo, partendo dai concetti generali fino alla comprensione completa dei dettagli operativi e dei rischi, facilitando l'apprendimento teorico e pratico dell'intero ecosistema Arch Linux e AUR.

2 Repository Ufficiali e Pacman

2.1 Come Funzionano i Repository Ufficiali

I repository ufficiali di Arch Linux sono strutture centralizzate contenenti pacchetti precompilati che vengono scaricati, verificati e installati da pacman. Questi pacchetti includono binari, librerie, documentazione e metadata dettagliati.

Il funzionamento avviene attraverso diversi passaggi:

- **Database dei repository:** pacman mantiene un database locale in `/var/lib/pacman/sync/` dei repository abilitati, contenente informazioni su ogni pacchetto: nome, versione, dipendenze, file inclusi, checksum e firme. Prima di installare, pacman aggiorna questo database con `pacman -Sy`.
- **Download del pacchetto:** durante `pacman -S <pacchetto>`, pacman scarica il pacchetto precompilato (.pkg.tar.zst) dal mirror specificato in `/etc/pacman.d/mirrorlist`, assicurandosi di rispettare la gerarchia dei server e le preferenze impostate.
- **Verifica dell'integrità e firma digitale:** pacman confronta il checksum (SHA256) del pacchetto con quello registrato nel database e verifica la firma digitale GPG per garantire autenticità e integrità.
- **Risoluzione delle dipendenze:** pacman analizza tutte le dipendenze dichiarate e le risolve ricorsivamente, scaricando automaticamente i pacchetti necessari se non presenti nel sistema.
- **Controllo dei conflitti e file ownership:** pacman verifica se i file da installare esistono già in altri pacchetti. In caso di conflitto, l'installazione viene bloccata, segnalando il pacchetto in conflitto e richiedendo l'intervento dell'utente.
- **Installazione nel filesystem:** i file del pacchetto vengono estratti nelle directory corrette (/usr/bin, /usr/lib, /usr/share, ecc.), preservando permessi, proprietari e attributi speciali.
- **Aggiornamento del database locale dei pacchetti:** pacman registra tutte le informazioni sul pacchetto installato in `/var/lib/pacman/local/<pkg>/`, comprese le liste dei file, versioni, trigger e install scripts, utili per future operazioni di upgrade o rimozione.
- **Trigger e script post-installazione:** pacman esegue eventuali script forniti nel pacchetto (ad esempio aggiornamento cache font, registrazione librerie in ldconfig, aggiornamento database MIME o GSettings) per integrare completamente il pacchetto nel sistema.

Questo processo garantisce che ogni pacchetto installato dai repository ufficiali sia sicuro, coerente e correttamente integrato nel sistema Arch Linux, minimizzando errori di dipendenza o conflitti tra pacchetti.

2.2 Installazione dei Pacchetti Precompilati

L'installazione dei pacchetti precompilati tramite `pacman` implica una serie di operazioni dettagliate per assicurare che il pacchetto sia sicuro, integro e correttamente posizionato nel filesystem. Questi passaggi includono sia controlli preliminari sia operazioni post-installazione:

- **Risoluzione del nome pacchetto:** pacman controlla il database locale per verificare l'esistenza del pacchetto richiesto e identifica la versione più recente disponibile nel repository.
- **Download del pacchetto:** il pacchetto precompilato (.pkg.tar.zst) viene scaricato dal mirror configurato, usando protocolli sicuri (HTTP/S) e rispettando la gerarchia dei mirror e le opzioni di fallback.
- **Verifica dell'integrità:** pacman confronta il checksum del pacchetto con quello presente nel database per assicurarsi che il file non sia corrotto o modificato.
- **Verifica della firma digitale:** se il pacchetto o il database sono firmati, pacman verifica la firma GPG per autenticare la provenienza del pacchetto.
- **Risoluzione delle dipendenze:** pacman analizza le dipendenze dichiarate nel pacchetto e, se non presenti nel sistema, le scarica e installa automaticamente. Questo include anche la gestione delle dipendenze ricorsive.
- **Controllo dei conflitti:** pacman verifica se l'installazione del pacchetto andrebbe a sovrascrivere file appartenenti ad altri pacchetti, prevenendo errori e garantendo integrità del filesystem.
- **Estrazione dei file:** il pacchetto viene estratto nelle directory appropriate (/usr/bin, /usr/lib, /usr/share, ecc.), preservando permessi, proprietari e attributi speciali dei file.
- **Aggiornamento del database locale:** pacman registra tutte le informazioni sul pacchetto appena installato nel database locale (`/var/lib/pacman/local/<pkg>/`) per gestire aggiornamenti futuri, rimozioni e trigger.
- **Esecuzione dei trigger e script post-installazione:** vengono eseguiti eventuali script forniti dal pacchetto (come aggiornamento cache font, ldconfig, database MIME o GSettings), completando l'integrazione del pacchetto nel sistema.

Questa sequenza garantisce che l'installazione dei pacchetti precompilati sia sicura, coerente con lo stato del sistema e completamente integrata con gli altri pacchetti presenti, minimizzando conflitti e errori di dipendenza.

2.3 Vantaggi e Limitazioni

L'utilizzo dei pacchetti precompilati dai repository ufficiali di Arch Linux presenta numerosi vantaggi, ma anche alcune limitazioni che è importante conoscere per una gestione efficace del sistema.

Vantaggi:

- **Stabilità e affidabilità:** i pacchetti nei repository ufficiali sono testati e curati dai manutentori Arch, riducendo il rischio di bug critici o incompatibilità.
- **Velocità di installazione:** essendo precompilati, i pacchetti possono essere installati rapidamente senza bisogno di compilazione locale.
- **Gestione automatica delle dipendenze:** pacman risolve automaticamente dipendenze dirette e ricorsive, evitando conflitti e installazioni incomplete.
- **Sicurezza:** pacchetti e database sono firmati digitalmente; l'integrità e l'autenticità vengono verificate automaticamente da pacman.
- **Trigger e script integrati:** eventuali script post-installazione e trigger vengono eseguiti automaticamente, assicurando che il pacchetto sia completamente integrato nel sistema.

Limitazioni:

- **Versioni non sempre aggiornate all'ultimo commit upstream:** i pacchetti nei repository ufficiali seguono le release stabili; eventuali aggiornamenti più recenti presenti nel repository upstream non sono immediatamente disponibili.
- **Meno flessibilità:** non è possibile modificare facilmente le opzioni di compilazione o applicare patch personalizzate, a differenza dei pacchetti AUR che puoi buildare localmente.
- **Possibili conflitti con pacchetti -git o personalizzati:** installare un pacchetto precompilato che condivide file con un pacchetto AUR o compilato localmente richiede di risolvere conflitti manualmente.
- **Dipendenza da mirror:** il download dei pacchetti precompilati dipende dalla disponibilità e aggiornamento dei mirror; un mirror non aggiornato può fornire versioni obsolete.

Questa analisi aiuta a comprendere quando conviene utilizzare pacchetti precompilati e quando, invece, può essere più vantaggioso ricorrere alla build locale tramite AUR, in base a esigenze di aggiornamento, personalizzazione o compatibilità del sistema.

3 AUR (Arch User Repository)

3.1 Cos'è l'AUR

L'AUR (Arch User Repository) 'e un repository comunitario gestito dagli utenti di Arch Linux, contenente metadati e ricette per la costruzione dei pacchetti, ma non i pacchetti precompilati. La filosofia alla base dell'AUR 'e quella di consentire agli utenti di condividere e mantenere pacchetti che non sono inclusi nei repository ufficiali, fornendo un ecosistema flessibile e in continua evoluzione.

Caratteristiche principali dell'AUR:

- **PKGBUILD:** ogni pacchetto AUR contiene un file PKGBUILD, uno script Bash che descrive dettagliatamente come costruire il pacchetto. Definisce variabili come `pkgname`, `pkgver`, `pkgrel`, `source`, `depends`, e funzioni come `prepare()`, `build()` e `package()`.
- **.SRCINFO:** un file generato a partire dal PKGBUILD che contiene metadati semplificati, usato dall'interfaccia web dell'AUR e dagli helper per verificare aggiornamenti.
- **Assenza di pacchetti binari:** l'AUR fornisce solo le istruzioni per buildare i pacchetti. L'utente o l'helper AUR deve compilare localmente i pacchetti prima dell'installazione.
- **Variazioni di pacchetti:** l'AUR ospita pacchetti stabili e pacchetti -git che seguono i commit più recenti del repository upstream. I pacchetti -git hanno `pkgver()` dinamico e spesso source puntati a repository Git.
- **Helper AUR:** strumenti come `yay`, `paru`, `trizen` automatizzano la clonazione del PKGBUILD, la risoluzione delle dipendenze di build, l'esecuzione di `makepkg` e l'installazione finale con `pacman -U`.
- **Controllo delle dipendenze:** makepkg distingue tra `depends` (dipendenze runtime) e `makedepends` (necessarie solo per la build). Gli helper AUR installano automaticamente le makedepends mancanti.
- **Sicurezza e verifiche:** prima della build, 'e fondamentale ispezionare il PKGBUILD, verificare checksum, firme PGP e potenziali comandi pericolosi. L'AUR non ha garanzie di sicurezza come i repository ufficiali.
- **Conflitti e integrazione:** installare pacchetti AUR che sovrappongono file con pacchetti ufficiali o altri AUR richiede attenzione e gestione dei conflitti tramite `conflicts`, `provides` e `replaces`.
- **Vantaggi dell'AUR:** consente di accedere a software non incluso nei repo ufficiali, versioni più aggiornate, opzioni di build personalizzate e la possibilità di contribuire direttamente alla comunità.
- **Limitazioni dell'AUR:** richiede competenze tecniche maggiori, il processo di build pu'o fallire per dipendenze mancanti, versioni instabili o errori di compilazione; la sicurezza dipende dall'ispezione dell'utente.

L'AUR rappresenta quindi una risorsa potente ma che richiede consapevolezza: l'utente deve comprendere il processo di build, la struttura dei PKGBUILD e i rischi associati a sorgenti non verificate, garantendo una gestione consapevole e sicura dei pacchetti Arch Linux.

3.2 Differenze tra AUR e Repository Ufficiali

La principale differenza tra i repository ufficiali e l'AUR riguarda il tipo di pacchetti, il livello di sicurezza e il processo di installazione.

- **Tipologia di pacchetti:** nei repository ufficiali i pacchetti sono precompilati e pronti all'uso, mentre nell'AUR i pacchetti sono forniti come ricette (PKGBUILD) da compilare localmente.
- **Sicurezza e affidabilità:** i pacchetti ufficiali sono testati e firmati dai manutentori Arch, garantendo stabilit'a e integrit'a. Nell'AUR la sicurezza dipende dall'ispezione dell'utente: pacchetti malevoli o mal scritti possono introdurre rischi se non controllati.
- **Frequenza degli aggiornamenti:** i pacchetti ufficiali seguono le release stabili upstream, mentre l'AUR spesso ospita versioni pi'u recenti, incluso software -git con aggiornamenti continui dai repository upstream.
- **Personalizzazione:** nell'AUR 'e possibile modificare il PKGBUILD, cambiare opzioni di compilazione, patchare il codice sorgente o scegliere versioni alternative, cosa non possibile con pacchetti ufficiali precompilati.

- **Processo di installazione:** pacman gestisce l'installazione automatica, dipendenze e trigger per pacchetti ufficiali. Gli helper AUR (come yay) automatizzano la build e l'installazione dei pacchetti AUR, ma coinvolgono pi'u passaggi e richiedono l'esecuzione di script locali.
- **Conflitti e integrazione:** i pacchetti ufficiali sono progettati per integrarsi senza problemi nel sistema, mentre i pacchetti AUR possono confliggere con pacchetti ufficiali se non dichiarati correttamente in `conflicts` o `provides`.
- **Gestione delle dipendenze di build:** l'AUR distingue tra `depends` (runtime) e `makedepends` (necessarie solo per la build), mentre i repository ufficiali si occupano solo delle dipendenze runtime essenziali.

In sintesi, i repository ufficiali sono ideali per stabilit'a e semplicit'a, mentre l'AUR offre flessibilit'a, versioni aggiornate e possibilità di personalizzazione, a fronte di una maggiore responsabilit'a nella gestione della sicurezza e dei conflitti.

3.3 Ruolo degli Helper AUR (yay, paru, ecc.)

Gli helper AUR, come `yay`, `paru` e `trizen`, sono strumenti che automatizzano il processo di installazione dei pacchetti AUR. Essi non sostituiscono `pacman`, ma integrano la sua funzionalit'a con operazioni necessarie per buildare e installare pacchetti dall'AUR.

Funzioni principali degli helper AUR:

- **Clonazione dei PKGBUILD:** scaricano il PKGBUILD e gli eventuali file di supporto dal repository AUR, spesso tramite git o download diretto.
- **Risoluzione delle dipendenze di build:** identificano le `makedepends` mancanti e le installano automaticamente tramite pacman prima di avviare la build.
- **Esecuzione di makepkg:** eseguono `makepkg` per compilare localmente il pacchetto, gestendo opzioni come `-si` (installa dopo build) o `-c` (pulisce la directory build).
- **Installazione automatica:** una volta creato il pacchetto .pkg.tar.zst, l'helper chiama `pacman -U` per installarlo, gestendo aggiornamenti e conflitti.
- **Aggiornamenti:** gli helper possono controllare periodicamente la disponibilit'a di nuove versioni dei pacchetti AUR e facilitare l'aggiornamento automatico, integrando anche i pacchetti ufficiali nel medesimo flusso.
- **Funzionalit'a avanzate:** molti helper offrono ricerca interattiva, opzioni di filtro (ad esempio installare solo pacchetti -git, o escludere pacchetti instabili), gestione dei conflitti e caching dei PKGBUILD.
- **Sicurezza e consapevolezza:** nonostante l'automazione, l'utente deve sempre verificare i PKGBUILD prima della build; gli helper facilitano, ma non sostituiscono la responsabilit'a dell'utente nel controllare sorgenti e script.

In sostanza, gli helper AUR semplificano enormemente l'interazione con l'AUR, rendendo possibile l'installazione di pacchetti da sorgente quasi come fosse un pacchetto precompilato, pur mantenendo trasparenza e controllo sul processo di build.

4 Il PKGBUILD

4.1 Struttura Generale del File

Il PKGBUILD 'e uno script Bash che descrive dettagliatamente come costruire un pacchetto Arch Linux a partire dai sorgenti. "Buildare" in questo contesto significa compilare il codice sorgente o preparare i file binari, le librerie e la documentazione per creare un pacchetto pronto all'installazione (.pkg.tar.zst).

La struttura generale di un PKGBUILD comprende:

- **Variabili obbligatorie:**
  - `pkgname`: nome del pacchetto.
  - `pkgver`: versione del pacchetto.

  - `pkgrel`: numero di rilascio del pacchetto per la stessa versione upstream.
  - `arch`: architetture supportate (ad es. x86_64).
  - `url`: pagina ufficiale o upstream.
  - `license`: licenze del software.
  - `depends`: dipendenze necessarie a runtime.
  - `makedepends`: dipendenze necessarie solo per la build.
  - `source`: URL o path dei sorgenti.
  - `sha256sums`: checksum per verificare integrità dei sorgenti.

  “‘

- **Funzioni principali:**
  - `prepare()`: prepara il sorgente, applica patch o configurazioni necessarie prima della build.
  - `build()`: compila il codice sorgente o prepara i file binari/librerie/documentazione.
  - `check()`: opzionale, esegue test sul software compilato.
  - `package()`: copia i file generati nella struttura di destinazione `$pkgdir` rispettando la gerarchia standard (/usr/bin, /usr/lib, /usr/share).
  - `pkgver()`: opzionale, funzione dinamica per generare automaticamente la versione del pacchetto, tipica nei pacchetti -git.

  “‘

- **Commenti e metadati aggiuntivi:** descrizioni, archivi alternativi, patch da applicare, e qualsiasi altra informazione utile al build e all’integrazione nel sistema.

Il PKGBUILD ‘e quindi una "ricetta" completa: definisce dove prendere il codice sorgente, come compilarlo/buildarlo, come organizzare i file nel pacchetto e quali dipendenze soddisfare, consentendo a makepkg e agli helper AUR di creare pacchetti installabili in maniera ripetibile e sicura.

4.2 Variabili Principali

Le variabili principali di un PKGBUILD definiscono informazioni essenziali sul pacchetto e sul processo di build. Conoscere queste variabili ‘e fondamentale per capire come makepkg costruisce il pacchetto e come pacman lo installa.

- **pkgname**: il nome del pacchetto. Deve essere univoco all’interno del sistema e coerente con le convenzioni Arch.
- **pkgver**: la versione del software upstream. Utilizzata per confrontare aggiornamenti e versioni.
- **pkgrel**: il numero di rilascio del pacchetto Arch per una stessa versione upstream. Incrementato ogni volta che si modifica il PKGBUILD senza cambiare `pkgver`.
- **arch**: architetture supportate, ad esempio x86_64, i686. Imposta le architetture per cui il pacchetto ‘e costruito.
- **url**: URL del progetto o del sito ufficiale upstream.
- **license**: licenze del software, come GPL, MIT, BSD. Importante per conformit‘a legale.
- **depends**: elenco delle dipendenze richieste a runtime. Pacman le installer‘a se mancanti.
- **makedepends**: dipendenze richieste solo per costruire il pacchetto. Gli helper AUR le installeranno automaticamente prima della build.
- **source**: URL o path dei file sorgente da cui compilare o preparare il pacchetto.
- **sha256sums**: checksum SHA256 dei file sorgente per garantire integrit‘a e sicurezza.
- **groups**: opzionale, permette di raggruppare pacchetti in gruppi tematici (ad es. base-devel).
- **conflicts**: pacchetti con cui questo pacchetto entra in conflitto. Pacman bloccher‘a l’installazione se sono presenti.
- **provides**: pacchetti virtuali forniti dal pacchetto, utile per dipendenze alternative.
- **replaces**: pacchetti sostituiti dal pacchetto in questione.

Queste variabili, insieme alle funzioni di build, permettono a makepkg di costruire pacchetti coerenti e ripetibili, consentendo a pacman di gestirli correttamente nel sistema Arch Linux.

4.3 Funzioni build() e package()

Le funzioni `build()` e `package()` in un PKGBUILD sono fondamentali per la creazione dei pacchetti AUR, poich‘e definiscono come il software viene compilato e come i file vengono organizzati nel pacchetto finale.

- **build()**: questa funzione prende i file sorgente e li trasforma in binari o librerie pronti all’uso. In pratica, "buildare" significa:
  - Configurare l’ambiente di compilazione.
  - Eseguire il processo di compilazione o preparazione dei file (es. con `./configure && make` o strumenti equivalenti).
  - Applicare patch, configurazioni o script aggiuntivi necessari per adattare il software al sistema.

  La funzione `build()` non deve modificare direttamente il filesystem di sistema; tutti i file prodotti devono essere collocati nella directory temporanea gestita da makepkg.
  “‘
- **package()**: questa funzione si occupa di installare i file generati da `build()` nella struttura di destinazione del pacchetto, definita da `$pkgdir`.
  - Copia i binari in `/usr/bin`, librerie in `/usr/lib`, documentazione in `/usr/share/doc`, ecc.
  - Imposta permessi, proprietari e attributi corretti per ciascun file.
  - Include eventuali script o file di configurazione necessari all’integrazione del pacchetto nel sistema.
- **Separazione dei ruoli**: `build()` si occupa esclusivamente della compilazione e preparazione dei file, mentre `package()` organizza e crea la struttura definitiva del pacchetto installabile. “‘

Questa separazione garantisce che la build sia ripetibile e isolata, mentre la funzione package() consente a pacman di gestire correttamente l’installazione dei file nel sistema, preservando sicurezza e coerenza.

4.4 Gestione delle Dipendenze

La gestione delle dipendenze ‘e un aspetto fondamentale sia per i pacchetti dei repository ufficiali sia per i pacchetti AUR, poich‘e garantisce che il software funzioni correttamente una volta installato.

- **Dipendenze runtime (depends):** queste sono librerie o pacchetti necessari al funzionamento del software installato. Pacman e gli helper AUR verificano che siano presenti e le installano automaticamente se mancanti.
- **Dipendenze di build (makedepends):** necessarie solo durante il processo di compilazione dei pacchetti AUR. Esempi tipici includono compilatori, strumenti di sviluppo o librerie temporanee. Gli helper AUR le installano automaticamente prima di eseguire `makepkg`.
- **Dipendenze opzionali (optdepends):** pacchetti non essenziali ma che abilitano funzionalit‘a extra. La loro installazione ‘e suggerita ma non obbligatoria.
- **Risoluzione automatica:** pacman analizza ricorsivamente le dipendenze dei pacchetti e installa tutto il necessario. Gli helper AUR replicano questo comportamento anche per le `makedepends`.
- **Conflitti e virtual packages:** le variabili `conflicts`, `provides` e `replaces` gestiscono conflitti, sostituzioni e pacchetti virtuali. Questo permette di installare pacchetti alternativi senza problemi di compatibilit‘a.
- **Sicurezza e coerenza:** il controllo delle dipendenze evita installazioni incomplete o danneggiate, riducendo rischi di crash o malfunzionamenti.

Una corretta gestione delle dipendenze garantisce che sia i pacchetti precompilati sia i pacchetti AUR siano integrati nel sistema in maniera coerente, stabile e sicura, prevenendo errori durante l’esecuzione del software e semplificando aggiornamenti futuri.

4.5 Generazione della Versione tramite pkgver()

La funzione `pkgver()` in un PKGBUILD ‘e opzionale ma estremamente utile, specialmente per i pacchetti -git o per pacchetti che seguono una versione dinamica upstream. Questa funzione genera la versione del pacchetto basandosi sullo stato attuale del codice sorgente, consentendo di avere sempre numeri di versione aggiornati e coerenti.

- **Pacchetti stabili:** per pacchetti con versioni fisse, `pkgver()` di solito non viene definita, e il PKGBUILD usa la variabile `pkgver` statica.
- **Pacchetti -git:** questi pacchetti prendono il codice dall’ultimo commit di un repository git. La funzione `pkgver()` esegue comandi come:

  ```
  pkgver() {
  cd "$srcdir/$_pkgname"
  git describe --long --tags
  }
  ```

  Questo genera versioni del tipo `0.52.0.r1283.g4acbfe3`, dove:

  - `0.52.0` = ultimo tag upstream,
  - `r1283` = numero di commit dal tag,
  - `g4acbfe3` = hash del commit.
- **Vantaggi:** questa generazione dinamica permette di sapere esattamente quale versione del codice sorgente ‘e stata buildata, rendendo tracciabili aggiornamenti frequenti senza modificare manualmente `pkgver`.
- **Integrazione con makepkg:** durante la build, makepkg chiama automaticamente `pkgver()` se presente, sostituendo il valore della variabile `pkgver` statica con quello calcolato.
- **Best practice:** usare `pkgver()` solo quando necessario, verificare che non produca valori inconsistenti, e assicurarsi che il numero risultante segua la convenzione Arch per l’ordinamento delle versioni.

Questa funzione consente quindi di avere un versionamento preciso e automatico dei pacchetti derivati da repository in rapido sviluppo, come quelli -git, senza intervento manuale dell’utente.

5 Pacchetti -git

5.1 Caratteristiche dei Pacchetti Basati su Repository Git

I pacchetti basati su repository Git, spesso indicati con il suffisso -git nell’AUR, hanno caratteristiche particolari rispetto ai pacchetti stabili dei repository ufficiali.

- **Versione dinamica:** la versione del pacchetto viene generata automaticamente tramite la funzione `pkgver()`, riflettendo l’ultimo commit presente nel repository Git. Questo permette di avere sempre la versione pi‘u aggiornata del software.
- **Aggiornamenti frequenti:** essendo legati al repository upstream, i pacchetti -git ricevono aggiornamenti continui, anche più volte al giorno, diversamente dai pacchetti stabili che seguono le release ufficiali.
- **Compilazione locale obbligatoria:** a differenza dei pacchetti precompilati, i pacchetti -git richiedono la build locale tramite `makepkg` o helper AUR. Non esistono binari preconfezionati.
- **Controllo delle dipendenze di build:** le `makedepends` devono essere presenti per compilare correttamente il pacchetto. Helper come `yay` installano automaticamente queste dipendenze.
- **Rischio di instabilit‘a:** i pacchetti -git possono introdurre bug o comportamenti instabili poich‘e seguono lo stato corrente dello sviluppo, non una release testata.
- **Tracciabilit‘a dei commit:** la versione generata include il numero di commit e l’hash del commit, permettendo di identificare esattamente il codice sorgente usato per la build.

- **Integrazione con gli helper AUR:** helper come `yay` gestiscono automaticamente il pull dei repository Git, l'esecuzione di `makepkg`, e l'installazione dei pacchetti aggiornati.
- **Personalizzazione e patching:** essendo costruiti localmente, questi pacchetti consentono facilmente di applicare patch o modifiche personalizzate prima della build.

I pacchetti basati su Git offrono quindi l'accesso immediato all'ultima versione del software, massima flessibilit'a e trasparenza del codice sorgente, ma richiedono attenzione alla gestione delle dipendenze e alla stabilit'a del sistema.

5.2 Versionamento Automatico

Il versionamento automatico nei pacchetti AUR, in particolare nei pacchetti -git, 'e una caratteristica che consente di generare numeri di versione dinamici basati sullo stato del repository upstream. Questa funzionalit'a si integra con la funzione `pkgver()` per fornire un identificatore preciso della versione del software compilato.

- **Numeri di commit e hash:** la versione generata include l'ultimo tag upstream, il numero di commit successivi al tag e l'hash del commit, ad esempio `0.52.0.r1283.g4acbfe3`.
- **Rilevanza per gli aggiornamenti:** il versionamento automatico permette agli helper AUR di determinare se esiste un nuovo commit da buildare, semplificando la gestione di pacchetti in rapido sviluppo.
- **Compatibilit'a con pacman:** il numero di versione dinamico rispetta la convenzione Arch per ordinamento e confronto delle versioni, garantendo che pacman possa gestire correttamente aggiornamenti e conflitti.
- **Vantaggi:** consente tracciabilit'a precisa, facilita l'integrazione con script di automazione, e riduce la necessit'a di modifiche manuali alla variabile `pkgver`.
- **Best practice:** utilizzare il versionamento automatico solo quando il pacchetto segue repository in rapido sviluppo, verificando che le versioni generate siano coerenti e ordinate correttamente.

Il versionamento automatico, combinato con i pacchetti -git e gli helper AUR, offre un sistema potente per mantenere i pacchetti sempre aggiornati, pur mantenendo coerenza e tracciabilit'a nella gestione delle versioni.

5.3 Differenze con i Pacchetti Stabili

I pacchetti -git o basati su repository Git si differenziano dai pacchetti stabili dei repository ufficiali su diversi aspetti fondamentali:

- **Aggiornamenti:** i pacchetti stabili seguono le release ufficiali upstream, mentre i pacchetti -git vengono aggiornati ad ogni nuovo commit, offrendo accesso immediato alle ultime modifiche.
- **Versionamento:** i pacchetti stabili hanno una versione fissa e un numero di release incrementale (pkgrel), mentre i pacchetti -git utilizzano versionamento dinamico tramite `pkgver()` con tag, numero di commit e hash.
- **Compilazione:** i pacchetti stabili sono generalmente precompilati nei repository ufficiali, mentre i pacchetti -git richiedono la build locale da sorgente.
- **Stabilità e rischio:** i pacchetti stabili sono testati e affidabili, mentre i pacchetti -git possono introdurre bug o comportamenti instabili poich'e seguono lo sviluppo in tempo reale.
- **Personalizzazione:** i pacchetti -git permettono modifiche al PKGBUILD, patch locali e opzioni di compilazione personalizzate, cosa che non 'e possibile con i pacchetti stabili precompilati.
- **Tracciabilit'a:** i pacchetti -git includono informazioni sui commit esatti utilizzati per la build, rendendo facile identificare lo stato esatto del codice sorgente, mentre i pacchetti stabili riportano solo la versione ufficiale.
- **Dipendenze:** i pacchetti stabili gestiscono solo le dipendenze runtime, mentre i pacchetti -git richiedono anche makedepends per la build, con installazione automatica da parte degli helper AUR.

Queste differenze evidenziano i compromessi tra stabilit'a e aggiornamenti rapidi: i pacchetti stabili sono ideali per sistemi affidabili e coerenti, mentre i pacchetti -git offrono le ultime funzionalit'a a costo di una maggiore attenzione alla gestione delle dipendenze e alla stabilit'a complessiva del sistema.

6 Conflitti tra Pacchetti

6.1 Come Nascono i File Conflicts

I file conflicts nascono quando due pacchetti tentano di installare nello stesso percorso file con lo stesso nome, causando sovrascritture indesiderate. Questa situazione pu'o verificarsi sia tra pacchetti dei repository ufficiali sia tra pacchetti AUR o tra pacchetti ufficiali e AUR.

- **File duplicati:** se due pacchetti includono lo stesso binario, libreria o file di configurazione, pacman rileva il conflitto e blocca l'installazione per evitare corruzioni del sistema.
- **Pacchetti -git e stabili:** spesso pacchetti -git installano file identici a quelli presenti nei pacchetti stabili, causando conflitti se non rimossi o gestiti correttamente.
- **Sovrapposizione di directory:** non solo i file singoli, ma anche directory condivise possono causare conflitti se i permessi o i contenuti non sono coerenti.
- **Variabili PKGBUILD coinvolte:** le direttive `conflicts`, `provides` e `replaces` aiutano a dichiarare e risolvere preventivamente conflitti tra pacchetti.
- **Gestione automatica:** pacman utilizza queste informazioni per bloccare o permettere installazioni in base a conflitti dichiarati. Gli helper AUR rispettano la stessa logica durante l'installazione dei pacchetti buildati localmente.

Comprendere come nascono i conflitti 'e essenziale per gestire correttamente l'interazione tra pacchetti ufficiali e pacchetti AUR, evitando sovrascritture e garantendo integrit'a del filesystem.

6.2 Perché hyprland e hyprland-git Confliggono

Il conflitto tra `hyprland` (dal repository ufficiale) e `hyprland-git` (dall'AUR) nasce principalmente dalla sovrapposizione dei file installati e dalla gestione delle versioni diverse:

- **File identici:** entrambi i pacchetti installano file binari in `/usr/bin/Hyprland`, librerie in `/usr/lib/libhyprland*` e risorse in `/usr/share/hyprland/`. La presenza simultanea provoca file conflicts.
- **Versioni differenti:** `hyprland` dal repository ufficiale 'e stabile e ha versioni numeriche definite, mentre `hyprland-git` segue l'ultimo commit upstream, generando versioni dinamiche (es. 0.52.0.r1283.g4acbfe3) tramite `pkgver()`.
- **Gestione pacchetti da pacman:** pacman rileva che due pacchetti vogliono scrivere negli stessi percorsi e blocca l'installazione, evitando corruzioni del filesystem.
- **Dipendenze e coerenza:** installare entrambi potrebbe causare incompatibilità tra librerie o comportamenti inattesi del software, poich'e il sistema non sa quale versione considerare primaria.
- **Risoluzione dichiarata:** per prevenire conflitti, i PKGBUILD possono dichiarare `conflicts` con pacchetti specifici, ma in assenza di dichiarazione, pacman rileva comunque la sovrapposizione di file e blocca l'operazione.

Questa comprensione 'e essenziale per chi vuole gestire hyprland in versione stabile o -git, poich'e installare uno richiede la rimozione o il backup dell'altro per evitare conflitti.

6.3 Strategie di Risoluzione

Per gestire i conflitti tra pacchetti come `hyprland` e `hyprland-git`, esistono diverse strategie per garantire la corretta installazione senza corrompere il sistema.

- **Disinstallare il pacchetto in conflitto:** prima di installare `hyprland-git`, rimuovere `hyprland` con `sudo pacman -R hyprland`. Questo elimina i file duplicati e previene conflitti.

- **Backup dei file personalizzati:** se sono presenti configurazioni o modifiche locali, salvarle prima di disinstallare il pacchetto in conflitto.
- **Utilizzare PKGBUILD con `conflicts` dichiarato:** alcuni pacchetti AUR dichiarano automaticamente i conflitti con pacchetti ufficiali, semplificando la gestione tramite pacman.
- **Gestione tramite helper AUR:** strumenti come `yay` rilevano automaticamente pacchetti in conflitto e suggeriscono la rimozione prima di procedere con la build e l'installazione.
- **Isolamento dei pacchetti (opzionale):** in casi avanzati, si possono installare pacchetti AUR in directory non standard o containerizzati per evitare sovrapposizioni con pacchetti ufficiali.
- **Aggiornamento ordinato:** aggiornare sempre prima i pacchetti ufficiali o AUR per evitare conflitti dovuti a versioni diverse di file condivisi.
- **Verifica post-installazione:** controllare che i file principali e le librerie siano coerenti e che il software funzioni correttamente dopo l'installazione.

Seguendo queste strategie, si pu'o installare e aggiornare pacchetti -git come `hyprland-git` senza interferire con i pacchetti stabili del repository ufficiale, mantenendo il sistema stabile e coerente.

7 Conclusioni

7.1 Riepilogo dei Concetti Chiave

In questa guida abbiamo esplorato in dettaglio l'ecosistema dei pacchetti Arch Linux e AUR, con particolare attenzione a pacchetti come `hyprland` e `hyprland-git`. Ecco i punti principali:

- **Repository ufficiali:** contengono pacchetti precompilati, stabili e firmati, gestiti da pacman con sicurezza e coerenza garantite.
- **AUR:** repository di pacchetti non precompilati, contenenti PKGBUILD che definiscono come buildare e installare il software localmente.
- **PKGBUILD:** file script che descrive sorgenti, dipendenze, funzioni di build e package, versionamento e gestione di conflitti.
- **Variabili principali:** `pkgname`, `pkgver`, `pkgrel`, `depends`, `makedepends`, `source`, `sha256sums`, `conflicts`, `provides` e `replaces`, che definiscono il comportamento del pacchetto.
- **Funzioni chiave:** `prepare()` per patch e preparazioni, `build()` per compilazione, `check()` per test opzionali, `package()` per organizzare file nel pacchetto, `pkgver()` per versionamento dinamico.
- **Helper AUR:** strumenti come `yay` automatizzano download, build, gestione delle dipendenze e installazione dei pacchetti AUR.
- **Pacchetti -git:** sempre aggiornati all'ultimo commit, richiedono build locale, versionamento dinamico e attenzione a stabilità e dipendenze.
- **Conflitti tra pacchetti:** nascono da file duplicati o directory sovrapposte; gestiti tramite `conflicts`, rimozione del pacchetto in conflitto e strategie sicure di aggiornamento.
- **Versionamento automatico:** fondamentale per pacchetti in sviluppo continuo, garantisce tracciabilità dei commit e corretto aggiornamento tramite helper.
- **Strategie di risoluzione:** disinstallazione dei pacchetti in conflitto, backup dei file, gestione tramite helper, aggiornamento ordinato e verifica post-installazione.

Questo riepilogo fornisce una base completa per comprendere come funziona Arch Linux nell'interazione tra pacman, AUR, PKGBUILD e pacchetti in rapido sviluppo come `hyprland-git`, consentendo una gestione consapevole e sicura del sistema.

7.2 Passi Successivi

Dopo aver compreso l'intero ecosistema dei pacchetti Arch Linux e AUR, e le differenze tra pacchetti stabili e -git, ecco i passi successivi consigliati per applicare le conoscenze:

- **Pratica con pacman:** installa, aggiorna e rimuovi pacchetti dai repository ufficiali per familiarizzare con la gestione base dei pacchetti.
- **Esplora l'AUR:** scegli un pacchetto semplice, scarica il PKGBUILD e analizzalo riga per riga per comprendere variabili e funzioni.
- **Usa un helper AUR:** prova con `yay` o `paru` per installare pacchetti AUR e osserva come vengono gestite dipendenze, build e installazione.
- **Crea o modifica un PKGBUILD:** prova a buildare un pacchetto personalizzato o a modificare un PKGBUILD esistente per capire il flusso completo.
- **Gestione dei conflitti:** esercitati con pacchetti in conflitto come `hyprland` e `hyprland-git` per applicare le strategie di risoluzione in sicurezza.
- **Versionamento e -git:** osserva come `pkgver()` genera versioni dinamiche, e come questo influisce su aggiornamenti e tracciabilità dei pacchetti.
- **Documentazione continua:** tieni traccia delle tue modifiche, delle versioni buildate e delle strategie adottate per avere un riferimento completo per futuri aggiornamenti.

Seguendo questi passi, sar'a possibile diventare autonomi nella gestione di pacchetti Arch Linux e AUR, comprendendo appieno cosa accade dietro le quinte durante la build e l'installazione dei pacchetti.