OrangeBook 2.0

Autistici/Inventati

Indice

1 Introduzione

Come funziona autistici.org? E soprattutto, perché funziona in un determinato modo? Lo scopo di questo documento è esaminare in dettaglio sia l'architettura tecnica che sta dietro ad Autistici/Inventati dal 2019, sia le scelte che l'hanno determinata.

Allo stesso tempo, oltre che la descrizione di uno stato di fatto, questa è anche una storia, un esempio crediamo positivo di una complessa transizione da un sistema *maturo* o *legacy*, con molto debito tecnico accumulato negli anni, ad un'architettura moderna, in grado di rispondere adeguatamente al problema che intende risolvere. In altre parole, senza toccare i motivi per cui A/I esiste, rivedere il *come* facciamo le cose nell'infrastruttura, *che cosa* ci tiriamo dietro da anni ed è modificabile, e *come* aggirare o convivere con l'approccio "mettici una pezza". Questo approccio fra riparazioni e workaround, porta tra l'altro ad un aumento della complessità conoscitiva, ad una conseguente fragilità tecnica e ad una maggiore difficoltà nell'introduzione di persone nuove che possano davvero mettere mano all'infrastruttura. Abbiamo cercato di realizzare un percorso sostenibile per le persone coinvolte, un aspetto fondamentale per un progetto di volontariato, provando ad introdurre alcuni selezionati concetti e alcune *best practices* provenienti dal mondo dell'industria del software, senza trasformarci in un contesto necessariamente di e per professionisti.

In questo libro siam costrette dalla lingua a declinare in qualche modo e quindi abbiam scelto di rivolgerci a noi stesse ed alle utenti al femminile, pensando a tutte le lotte che i trans/femminismi portano avanti e per supportarle. Nel testo usiamo Autistici/Inventati o A/I in riferimento al collettivo, ed autistici.org nei vari riferimenti tecnici per brevità, anche se l'infrastruttura è pensata per un utilizzo multi dominio: inventati.org, anche.no, noblogs.org, etc..

1.1 Contestualizzazione storica

1.1.1 *Piano R**

L'evoluzione tecnologica del progetto A/I ha seguito un percorso relativamente ortodosso per questi tipi di progetto, anticapitalista ed autogestito, passando rapidamente dal singolo server gestito con approccio "sperimentale", ai server multipli (in seguito alla crescita del numero di utenti ed a necessità geopolitiche) con la conseguente progressiva introduzione di vari meccanismi di *configuration management*. La specifica architettura dei servizi poi è mutata più volte nel tempo, in parallelo con il progresso della nostra comprensione

delle necessità strategiche del progetto stesso: per esempio, la struttura multi-livello discende direttamente dalla necessità di separare il "piano legale" (la superficie dell'architettura che è pubblicamente visibile e dunque soggetta a processi di *legal discovery*) dai dati delle utenti. Tale separazione è un aspetto che non desideriamo abbandonare, come neanche l'utilizzo di server dislocati in tutto il mondo, in una situazione in cui si continua a dare per scontata l'insicurezza fisica delle macchine, e rendendo quindi essenziale la progettazione di ciascuno dei "nodi" di questa rete come sostituibile.

Sul piano pratico, il progetto prevedeva l'acquisizione di hardware in differenti parti del mondo, per evitare di avere un unico punto di contatto commerciale su cui fare pressione. Questa scelta è stata fondamentale ed ha avuto ripercussioni su più livelli: innanzitutto ha reso necessaria la costituzione di una struttura organizzativa in grado di raccogliere fondi e pagare bollette regolarmente, tenere traccia dei contratti, etc. Questa attività è già, anche sulla nostra piccola scala, necessariamente oltre il livello amatoriale: non è semplicemente accettabile, per dire, perdere la possibilità di mandare mail perché questo mese si è dimenticato di pagare un server.

Inoltre la scelta di una distribuzione geografica mondiale ha determinato le scelte tecniche di implementazione discusse prima: un sistema distribuito su scala globale richiede tecnologie differenti da uno locale, essendo le latenze di trasmissione in gioco diverse di vari ordini di grandezza. In particolare abbiamo scelto di **non** implementare un layer di coordinamento globale, dividendo invece i servizi in componenti *stateless* (replicabili identicamente per ottenere high availability) e *stateful* (dove invece adottiamo il partizionamento come strategia di limitazione dei problemi), combinato con algoritmi di load balancing semplici e fondamentalmente *client-side* (round-robin DNS).

Sul piano tecnologico, l'evoluzione dell'infrastruttura negli anni ha portato dunque ad una soluzione a due livelli: un livello pubblico, composto da *reverse proxy* che ricevono il traffico utente e lo inoltrano al livello interno, che gestisce effettivamente i dati delle utenti. La comunicazione tra i due livelli avviene tramite una VPN. I dati delle utenti sono partizionati (ogni server contiene una "fetta" diversa dei dati), così da limitare eventuali problemi ad un sottoinsieme di loro. In questa architettura, le macchine pubbliche non contengono nessun dato importante, e sono facilmente sostituibili.

Questa struttura continua, nel 2022, a servire bene gli scopi che ci eravamo originariamente posti:

- Tutela dei dati delle utenti rispetto a vari tipi di abusi (sequestri, etc), costringendo le richieste legali a passare attraverso i canali ufficiali;
- un adeguato livello di affidabilità, corrispondente ad una situazione in cui problemi ad un singolo server sono o completamente invisibili (nel caso di un reverse proxy), oppure limitati ad un sottoinsieme delle utenti;
- la possibilità di crescere facilmente in risposta all'aumento di domanda, aggiungendo server (scalabilità cosiddetta *orizzontale*).

Rimandiamo all'OrangeBook originale (v1)[1] per avere più dettagli sull'implementazione e sulle scelte tecniche specifiche effettuate all'epoca. Architetture di questo tipo sono oggi molto comuni e necessitano di molte meno spiegazioni introduttive.

I presupposti identificati nel 2005 e sopra delineati rimangono dunque validi, e la struttura generale dei servizi continua a corrispondere grossomodo a quanto descritto. Recentemente però abbiamo introdotto dei cambiamenti drastici nel *modo* in cui pensiamo e gestiamo questa struttura. Nel corso di un paio di anni, fino alla sua inaugurazione nel 2019, abbiamo ripensato e riscritto completamente tutto il piano tecnologico di A/I, utilizzando l'esperienza di anni per immaginare una soluzione più adatta ai problemi attuali. Vediamo perché e come.

1.1.2 Stato tecnico di A/I ad inizio 2017

Come è normale per un lavoro prodotto da molte persone nel corso di molto tempo, la struttura tecnica ha finito con l'accumulare una serie di strati sedimentari storici, collegati da meccanismi imperfetti e *ad-hoc*. Se l'architettura di alto livello originale del 2005, il cosiddetto *Piano R**, che introduce i servizi anonimi, partizionati e delocalizzati, ci ha servito bene negli anni, la sua implementazione cominciava a mostrare preoccupanti segni di usura. I segni erano quelli caratteristici dell'accumulo di *technical debt*[2]:

- Lo sviluppo continuo, la lunga serie di miglioramenti del progetto nel tempo rende le vecchie soluzioni non ottimali;
- la presenza di molti "bug noti": parti dell'infrastruttura che non funzionano come dovrebbero, risultanti da implementazioni prototipali di meccanismi che poi non sono mai stati sostituiti da implementazioni migliori;
- la duplicazione di meccanismi per risolvere lo stesso problema, per esempio a seguito di migrazioni non completamente portate a termine;
- la difficoltà di risolvere un problema senza crearne uno nuovo, conseguente anche da un approccio vagamente ingenuo all'amministrazione di sistema che accentra servizi diversi sugli stessi demoni risultando in configurazioni estremamente complesse (un'unica istanza Postfix per i vari flussi di posta, un'unica istanza Apache per i vari siti web, etc) e molto difficili da testare;
- impossibilità di testare contemporaneamente su tutta l'infrastruttura, che incoraggia correzioni di bug rapide e/o rischiose.

Nel 2017 si verifica un evento che precipita la situazione: ci accorgiamo di un'intrusione in corso nei nostri sistemi, avvenuta grazie alla compromissione delle credenziali di un admin. L'analisi forense rivela che i meccanismi incompleti ed imperfetti precedentemente menzionati hanno avuto un ruolo nell'intrusione.

Un evento problematico di tale entità pone un problema esistenziale: anche avendo immediatamente risolto i vari singoli problemi specifici che hanno permesso l'intrusione, c'è fiducia nella capacità del collettivo di

[1] https://www.autistici.org/orangebook/
[2] https://en.wikipedia.org/wiki/Technical_debt

prevenire (non solo rimediare) eventi simili in futuro? Si intuisce che il problema è sistemico, e che all'origine di simili eventi ci sono proprio specifiche mancanze della attuale implementazione, mancanze che non sono state risolte proprio per via del *technical debt* descritto in precedenza.

Questo documento è la descrizione delle soluzioni che abbiamo elaborato, una volta deciso che valeva la pena continuare ad investire energie nel progetto, sia per mitigare i suddetti problemi, sia per evitare per quanto possibile che si ripetano in futuro.

2 Principi organizzativi

Quando ci siamo sedute assieme e più o meno collettivamente abbiamo considerato il da farsi, abbiamo cercato di immaginare contemporaneamente lo stato finale desiderato, ed un percorso possibile per arrivarci. Il nostro ragionamento si è basato su alcuni principi generali che è utile esaminare.

2.1 Sul piano umano e relazionale

Come strutturare un progetto in modo che possa sopravvivere sul lungo periodo? Un fattore critico, in un progetto di volontariato, è sempre quello umano, dunque è necessario plasmare l'aspetto tecnico per venire incontro agli umani che dovranno occuparsene.

L'organizzazione di A/I non è fatta solo da persone tecniche dell'informatica, sistemiste o programmatrici: per sviluppare un progetto di comunicazione resistente con solidi principi antagonisti è necessaria una componente umana, composta da persone che seguono l'ambito legale, economico, fiscale, comunicativo, politico e noiosissimamente burocratico, che ci permettono di essere più robuste nel momento in cui le nostre azioni, ma molto più spesso chi ne usufruisce o i contenuti ospitati, vengono messe in discussione. Coloro che tecnicamente scrivono codice, revisionano quello dell'*upstream*, stilano l'aspetto dei nostri servizi e studiano la sicurezza della nostra architettura, sono semplicemente una parte del gruppo. C'è inoltre chi di noi segue le richieste dei servizi e di supporto e alcuni ruoli inoltre per forza di cose si trovano a spaziare trasversalmente in tutti gli ambiti rendendo ancora più complesso il tutto.

Abbiamo dunque attribuito un ruolo primario, nelle nostre valutazioni, al tempo e all'attenzione delle admin che dovranno avere a che fare con il sistema, articolando i seguenti principi:

- avere sempre in mente, a fronte di qualunque scelta, la difficoltà cognitiva che questa comporta:
  - eliminare ad ogni costo la complessità evitabile
  - affrontare la complessità inevitabile rendendola almeno trattabile, fornendo il contesto delle scelte effettuate, aggiungendo documentazione ad ogni livello, costruendo sistemi che privilegino la chiarezza delle loro relazioni
- poter spiegare ed affidare l'infrastruttura a più persone con necessità di accomodare vari livelli di specializzazione e disponibilità, e supportare *engagement* a livello dei singoli "servizi" (es. "mi occupo solo della posta")

  - da qui l'esigenza di un'architettura modulare i cui vari componenti siano isolati tra loro. Condividere principi ed infrastruttura comuni massimizza la probabilità che, a fronte di un problema futuro, qualcuno abbia la possibilità di individuarlo e risolverlo

- necessità di mandare avanti la baracca con il minore sforzo possibile, consci delle difficoltà di ricambio umano e di disponibilità di tempo

  - da qui, oltre che al semplice fattore di scala, la scelta di costruire automazione, e sistemi distribuiti "resilienti": quando si pianificano architetture su scala decennale, l'investimento iniziale tende ad essere giustificato dal successivo minore costo operativo (sia in termini di tempo sia in termini di stress/urgenza)

- avere un piano più ampio di condivisione rispetto al design di una infrastruttura, che a parte le credenziali di amministrazione, è in questo modo visibile, comprensibile, usabile e migliorabile anche da altri gruppi.

2.2 Sul piano tecnico

Dai parametri delineati qui sopra discendono alcune scelte fondamentali sul piano tecnico che definiscono il tipo di soluzione che stavamo cercando:

- **Infrastructure-as-code** ovvero la decisione di controllare i sistemi descrivendoli come *codice* (configurazioni). Questa scelta ha conseguenze fondamentali sia sul piano tecnico sia sul piano sociale. Innanzitutto questo per noi è un principio di trasparenza, tutto quello che succede è visibile da tutti e – in linea di principio – documentato, un antidoto alla formazione di conoscenze privilegiate da parte di chi ha più tempo. Nei gruppi disomogenei c'è una tendenza al consolidamento della consapevolezza situazionale (la conoscenza dei motivi per lo stato attuale delle cose) in sottogruppi ristretti, definiti dalla maggiore co-presenza online, a discapito di tutti gli altri. Forzare invece lo stato del sistema in una unica locazione aiuta a disinnescare questi meccanismi nefasti.

- Nel momento in cui tutto lo stato dei sistemi è rappresentabile come codice, si può adottare uno dei molti **workflow collaborativi** per lavorare assieme sul codice, facilitando la collaborazione all'interno di gruppi distribuiti. Alcuni sistemi che offrono queste possibilità sono oggi molto diffusi (es. GitHub), cosa che facilita di molto la partecipazione in quanto è probabile che le persone abbiano già familiarità con essi.

- Necessità di astrazioni adeguate a manipolare e comprendere i servizi implementati: in un contesto di *infrastructure-as-code*, cosa è esattamente questo codice, e come ci assicuriamo che sia comprensibile e manipolabile con facilità? Un insieme scoordinato di file di configurazione non è immediatamente comprensibile come un "servizio", serve invece allineare la *superficie di controllo* con il modello mentale che ci costruiamo. Questo è tradizionalmente il dominio degli strumenti di *configuration*

management, meccanismi tecnici di astrazione che colmano il divario tra il livello implementativo (es. la configurazione di una specifica istanza di un web server) ed il livello semantico (es. il fatto che questa specifica istanza di un web server faccia parte del servizio X).

- La **replicabilità** dell'infrastruttura come ambiente di test, per consentire di compiere esperimenti o verificare ipotesi in sicurezza, ed acquisire fiducia nelle proprie modifiche e capacità. Questo consente al gruppo di includere nuove persone e dargli mano libera per testare e proporre modifiche, senza mettere a repentaglio la stabilità dell'intero sistema.
- Dal punto precedente discende anche la **rigenerabilità** completa e separata dei servizi, ripartendo soltanto dal codice dei repository, sia durante gli aggiornamenti di routine sia in caso di compromissione o altri eventi catastrofici.
- Vogliamo infine ottenere una efficace **separazione** dei servizi, di modo che siano modularizzati e compartimentati. La modularizzazione favorisce il riutilizzo dei componenti e dunque aumenta la comprensibilità del sistema, la compartimentazione rende possibile lo sviluppo incrementale. Una robusta separazione dei servizi è anche il fondamento del modello di sicurezza dell'infrastruttura, indipendentemente dagli specifici meccanismi adottati.

Queste necessità ci hanno spinto ad una scelta di versionamento con git, una distribuzione delle configurazioni di tipo push, realizzata con Ansible, e una compartimentazione dei servizi attraverso *container* piuttosto che macchine virtuali.

Siamo consapevoli che molte di queste scelte si sovrappongono, oggidì, con le prassi del mondo dell'informatica come professione: più che essere un segno di supporto ideologico, questa sovrapposizione è da interpretarsi come manifestazione di una convergenza pragmatica di interessi. In ogni caso ridurre le barriere tra le prassi del collettivo e quelle "professionali", ove necessario, aumenta le possibilità che le persone possano formarsi esternamente al collettivo.

3 Ridefinire l'infrastruttura

3.1 Il substrato tecnologico

Le considerazioni nella sezione precedente determinano le scelte tecniche che abbiamo adottato. Dovendo pensare a qualcosa che potesse durare nel tempo, in particolare abbiamo deciso di focalizzare i nostri sforzi nel rendere l'architettura funzionale alla sua comprensione.

Abbiamo quindi cercato un'astrazione di alto livello di *servizi* che risolvano ciascuno un problema specifico usando un'*infrastruttura* di funzionalità comuni, partendo dai seguenti princìpi:

- una separazione chiara dei vari servizi tra loro, per rendere possibili modifiche indipendenti ed isolate;
- l'infrastruttura deve definire soltanto delle interfacce, così che le varie parti della sua implementazione possano essere modificate nel tempo senza effetti collaterali indesiderati.

Posto che si facciano le scelte giuste su come definire le suddette interfacce e superfici di controllo (che non è un problema da poco!), il modello descritto consente di minimizzare il lavoro strettamente necessario per la manutenzione a lungo termine riducendolo agli eventuali aggiornamenti necessari per tenere l'infrastruttura al passo coi tempi.

In qualunque contesto che riguardi del software c'è, sullo sfondo, un generatore di lavoro sotto forma dei necessari aggiornamenti di sicurezza e *bug fix*. L'isolamento dei servizi ci permette di distribuire questo lavoro nel tempo: anche eventi come l'avanzamento della versione *stable* di Debian diventano così gestibili senza troppa fatica.

3.2 Scegliere un "livello di servizio"

Prima di prendere ulteriori decisioni tecniche è importante chiarire quale livello di servizio si desidera ottenere. Anche nel caso di un progetto di volontariato come il nostro, dove è chiaro per le utenti che non si sta offrendo un servizio misurabile secondo gli stessi rigidi parametri quantitativi di un'entità professionale, è comunque importante avere un obiettivo, per capire quali e quanti sforzi siano giustificati.

Prendiamo l'esempio dei nostri servizi di posta. Trattandosi di comunicazione, in realtà il livello di servizio necessario è elevato, perché la posta elettronica è ancora centrale nella vita online delle persone, dunque

l'obiettivo qui è offrire il massimo livello di continuità possibile al servizio. Nello specifico ci interessa dunque un servizio distribuito con sufficiente replicazione, robusto rispetto al fallimento di singoli server (il problema operativo più comune che riscontriamo), almeno per quanto riguarda le funzionalità basilari di invio e lettura della posta. Guardando in dettaglio il servizio di posta però vi sono svariate componenti che non necessitano di questo livello di sforzo, dato che la funzionalità principale del servizio può comunque essere svolta, seppure eventualmente in modo degradato, anche se alcune componenti "secondarie" falliscono.

In generale, esaminando le caratteristiche dei servizi che offriamo, abbiamo adottato due modelli di servizio separati, corrispondenti a livelli diversi di criticità e qualità del servizio:

- Per i servizi *critici* si sono scelti modelli distribuiti con replicazione e ridondanza (le specifiche dipendono dal servizio, dalla replicazione pura per i servizi *stateless*, a meccanismi di *leader election*, etc); questo ci permette di offrire un livello di servizio aggregato migliore, in genere, di quello offerto dai singoli provider che utilizziamo.
- Per i servizi non critici abbiamo adottato un modello di servizi ad istanza singola, con un ripiego o *failover* gestito manualmente (eventuali dati associati sono automaticamente recuperati dai backup dall'automazione, è l'operazione di failover che va attivata manualmente). Oltre a servizi *stateless*, abbiamo anche innumerevoli servizi ausiliari che raccolgono dati di vario tipo, per cui però è perfettamente accettabile perdere un giorno di dati (per esempio perché vengono rigenerati periodicamente, etc): in questi casi la scelta di non implementare meccanismi di replicazione può semplificare notevolmente l'implementazione.

3.3 Un modello ideale di servizio

Visto l'insieme dei criteri delineati in precedenza, ed esaminata la struttura dei servizi che offriamo e il modo in cui si sono evoluti nel corso di vent'anni, ci siamo fatti un'idea del tipo di astrazioni che idealmente ci serve manipolare:

- Un servizio è composto da un insieme di software ben definito, dalle relative configurazioni, ed infine dai dati associati. Nel nostro caso la demarcazione tra i primi due elementi ed il terzo è netta: i meccanismi di automazione si occupano solo del primo gruppo, e non toccano *mai* i dati degli utenti.
- Pensiamo ad un servizio come ad un costrutto di alto livello ("la posta"), esso sarà poi costituito da molteplici componenti, *demoni*, etc. Questa struttura gerarchica è essenziale per mantenere comprensibile la rete delle relazioni tra i vari componenti e servizi, che deve in ogni caso entrare nella testa di una persona.
- Ci interessa gestire la presenza *pubblica* di un servizio separatamente dal servizio stesso. Questo perché la presenza pubblica (che comprende cose come i record DNS, certificati, etc) è il perimetro su cui si articola la nostra interfaccia legale, e vogliamo essere in grado di gestirlo indipendentemente dall'implementazione dei servizi stessi.

- Esistono servizi pubblici e servizi "non pubblici". Lo scopo dei secondi è in dipendenza ai primi, implementando interfacce interne a funzionalità comuni, laddove abbia senso farlo. La determinazione di questa soglia è un parametro critico nella gestione della complessità finale del sistema nel suo insieme: all'atto pratico, la maggioranza dei servizi "non pubblici" che abbiamo creato (come Pannello, Account Admin, Services) si colloca nell'area più sofisticata, ovvero la gestione delle utenze.
- Abbiamo requisiti piuttosto laschi sull'assegnazione specifica dei servizi alle macchine: ci interessano la possibilità di definire gruppi (per avere tra loro affinità d'accesso ai dati delle utenti), e la possibilità di applicare criteri globali come "devono esistere N istanze di questo servizio".

Descrivere servizi in questo modo ci consente di averne una rappresentazione completamente separata dall'hardware, cosa che consente eventualmente di trattare l'hardware come una pura *commodity*, potendone dunque ragionare sul piano meramente quantitativo. Questa prospettiva si allinea bene con il nostro modello di gestione dell'hardware, che noleggiamo da provider commerciali, strategia mutuata dal primo piano R*.

4 Implementazione

4.1 Sistema operativo distribuito

Scegliendo di adottare un modello di servizi distribuiti, viene naturale considerare come raggruppare funzionalità comuni in altri servizi, ed eventualmente si realizza che alcuni sono così comuni e generici, e contemporaneamente legati alla possibilità stessa di gestire servizi distribuiti, da costituire quello che si potrebbe definire come un *sistema operativo distribuito*.

Un altro modo di vedere la questione è considerare quali funzionalità sarebbero utili dal punto di vista di chi deve implementare un servizio. Alcuni esempi: il mio servizio probabilmente genera log, mi interessa che questi siano in qualche modo gestiti e resi visibili senza dover implementare questa cosa a mano separatamente per ciascun servizio. Oppure: il mio servizio deve parlare con altri servizi interni, mi interessa poterlo fare in modo sicuro senza dover implementare un meccanismo specifico per ciascuna coppia di servizi comunicanti.

Questi esempi descrivono *funzionalità orizzontali*, potenzialmente comuni a tutti i servizi. Nel loro insieme, queste costituiscono l'interfaccia del nostro "sistema operativo distribuito", che altrove nel testo, dove il contesto non si presta ad ambiguità, chiameremo semplicemente *infrastruttura*.

L'elenco completo include:

- un modo per portare codice e configurazioni sugli host
- un meccanismo per assegnare servizi ai vari host (*scheduler*)
- separazione UNIX dei servizi (un utente diverso per ogni servizio)
- raccolta automatica dei log e la loro gestione in un sistema centralizzato
- raccolta automatica di metriche per il monitoraggio
- un meccanismo interno per consentire ai servizi di trovarsi l'un l'altro (*service discovery*)
- un meccanismo interno per consentire ai servizi di parlarsi l'un l'altro in modo sicuro (nel nostro caso abbiamo scelto mTLS)
- la possibilità per i servizi di ricevere traffico da Internet, gestendo automaticamente tutto ciò che questo comporta (DNS, SSL, etc)

4.2 Un ponte tra *Configuration Management* e *Container Orchestration*

La combinazione del desiderio di compartimentazione dei servizi e della necessità di una descrizione più sistematica degli stessi ci hanno spinto verso le moderne soluzioni di *container orchestration*[1]. Nel momento in cui abbiamo elaborato queste riflessioni, la container orchestration stava iniziando a diventare popolare nell'industria grazie al progetto *Kubernetes*.

E qui immediatamente si riscontra un problema: le competenze tecniche del collettivo si collocavano all'epoca prevalentemente nell'ambito dell'amministrazione di sistema "convenzionale". Inoltre l'esperienza ci insegna che i cambiamenti di paradigma tecnologico sono estremamente traumatici per i gruppi volontari come il nostro, contribuendo ad esasperare le disomogeneità interne in termini di tempo, apprendimento, soddisfazione e capacità tecniche.

Abbiamo dunque deciso di costruire un percorso di apprendimento collettivo, che potesse traghettare il gruppo nel suo complesso dai vecchi paradigmi ai nuovi. Questo percorso collettivo adotta un approccio incrementale, introducendo elementi fondamentali della container orchestration e concetti di sistema operativo distribuito all'interno di un contesto familiare di *configuration management*[2].

Non esistendo un prodotto che soddisfasse i nostri requisiti, in particolare qualcosa che offrisse la possibilità di limitare la curva di apprendimento, abbiamo dovuto scriverne uno, decisione sofferta ma che ci è apparsa inevitabile a queste condizioni. Il risultato è float[3].

Float è, in poche parole, un sistema (semplice) di *service orchestration* costruito attorno ad Ansible, un software di *configuration management* piuttosto popolare.

Ci sono anzitutto alcuni commenti da fare su Ansible. Lo abbiamo scelto principalmente per la sua *banalità*: al di là delle sue varie idiosincrasie, è fondamentalmente un esecutore di semplici task sequenziali, non tanto diverso da uno shell script. Ciò lo rende molto simile al nostro precedente sistema di gestione delle configurazioni, tanto che riscrivere la maggior parte dei ruoli è stato un compito relativamente semplice. L'altra caratteristica importante di Ansible è che segue uno schema *push*, in cui le modifiche vengono applicate dal computer dell'admin che poi si connette ai vari host. Questo è l'opposto dello schema *pull* che avevamo adottato fino ad ora, ed è una scelta esplicita in risposta all'adattamento del nostro *threat model* rispetto alle possibilità di intrusione.

Abbiamo scelto esplicitamente di considerare float uno strumento generico, utilizzabile senza bisogno di comprenderne appieno l'implementazione, come qualunque altro tool open-source, ma capendo cosa fa. È stato investito molto tempo, di conseguenza, non solo per mantenerlo minimale ma per scriverne una documentazione chiara, realizzare tutorial e guide passo-passo, e quant'altro fosse necessario per garantire la formazione interna sullo strumento.

[1] https://en.wikipedia.org/wiki/Orchestration_(computing)
[2] https://it.wikipedia.org/wiki/Gestione_della_configurazione
[3] https://git.autistici.org/ai3/float

4.2.1 *Float*

Lo scopo di *float* è implementare il "sistema operativo distribuito" descritto sopra: float assume il controllo dell'hardware *bare metal* e lo trasforma in una struttura in grado di ospitare i servizi. Il principio fondamentale seguito durante lo sviluppo è stato quello di fare il minimo possibile per implementare le interfacce infrastrutturali, nel modo più semplice possibile: questo allo scopo sia di ridurre il tempo di sviluppo, che di mantenere la comprensibilità del risultato finale. Questo spiega perché float nel complesso abbia caratteristiche descrivibili come "efficaci ma rozze".

Float combina una descrizione di alto livello dei servizi con la possibilità di usare Ansible per gestirne la configurazione. In particolare, float è molto tollerante nella sua definizione di "servizio", e ci consente di includervi dei container come anche delle normali unit systemd (in questo caso presumibilmente installate via Ansible). Questa flessibilità è stata **fondamentale** per poter gestire la migrazione dei nostri servizi, permettendoci di trasformarli in container un po' per volta al ritmo che ci è stato possibile.

Un'altra caratteristica importante di float, forse quella che più di tutte lo rende un oggetto nonostante tutto *semplice*, è il fatto che opera uno scheduler *offline*. Al contrario di altri sistemi più evoluti, non esiste un componente online che per esempio sposti servizi da un host all'altro a seguito di problemi con l'hardware!

Questa può sembrare una grande limitazione, ma in realtà non preclude la realizzazione di servizi con *high availability*: semplicemente sposta la responsabilità sul servizio stesso. Float non può fare molto per un servizio di cui si abbia una sola istanza su un unico host, ma avendone istanze multiple (per definizione su host differenti) il servizio può rimanere in grado di operare a fronte del fallimento di una singola istanza, fallimento di cui preferiamo esserne informate piuttosto che no. Lo scheduler *offline* impedisce di applicare anche altri pattern popolari nel mondo della container orchestration come ad esempio l'*autoscaling*, ovvero la modifica dinamica del numero di repliche di un servizio a seconda del suo utilizzo, ma queste non sono features che ci interessano essendo il nostro traffico nel complesso molto prevedibile.

Nello specifico, float è configurato mediante un file YAML, tradizionalmente chiamato `services.yml`, contenente la descrizione di alto livello dei servizi. La descrizione di ciascun servizio, oltre al nome che lo contraddistingue univocamente, specifica quali container devono essere configurati, quali porte di rete vanno esposte, e vari altri metadati relativi ai servizi infrastrutturali. Un esempio:

```yaml
archive:
  num_instances: 3
  containers:
    - name: http
      image: registry.git.autistici.org/apache
      env:
        APACHE_PORT: 8080
      volumes:
        - /var/lib/archive: /data
```

```
    port: 8080
  public_endpoint:
    name: archive
    port: 8080
```

Questo frammento di YAML descrive un ipotetico servizio *archive*, costituito dall'immagine di container *registry.git.autistici.org/apache*, cui diciamo di mettersi in ascolto sulla porta 8080 attraverso la variabile di environment `APACHE_PORT` (assumendo che questo sia il modo in cui questa immagine accetta la propria configurazione). La directory (sul server) */var/lib/archive* apparirà, dentro il container, come */data* (di nuovo, assumendo che sia lì che l'immagine si aspetti di trovare i propri dati).

La specifica impone a float di eseguire 3 istanze differenti di questo servizio, che dunque verranno distribuite su 3 server differenti. Altri servizi potranno connettersi a queste istanze, internamente, risolvendo il nome del servizio ("archive") tramite DNS.

Inoltre, attraverso la definizione di un *public_endpoint*, stiamo richiedendo a float di esporre questo servizio pubblicamente via HTTPS (come un sito chiamato a sua volta *archive*), al che float si occuperà di generare gli appositi record DNS ed ottenere dei certificati SSL validi, nonché delle configurazioni varie per il reverse proxy HTTP, etc.

A questa descrizione si può abbinare un ruolo Ansible, quando ciò sia necessario, per svolgere compiti che non rientrano nelle possibilità offerte dai metadati. Supponiamo che il nostro servizio ipotetico abbia bisogno di un particolare file, avremo dunque un ruolo Ansible *archive* con i seguenti task:

```
- name: Create archive data directory
  file:
    path: /var/lib/archive
    state: directory
    owner: docker-archive
- name: Create archive index page
  copy:
    dest: /var/lib/archive/index.html
    content: "hello world"
```

Per illustrare l'indifferenza dello strumento rispetto al dualismo container / servizio tradizionale, consideriamo invece una implementazione alternativa di questo ipotetico servizio che, anziché utilizzare dei container, fa uso di un pacchetto Debian (un egualmente ipotetico pacchetto "archive-server") con un'associata unit systemd. La descrizione potrebbe dunque diventare:

```
archive:
  num_instances: 3
  systemd_services:
    - archive-server
  public_endpoint:
```

```
    name: archive
    port: 8080
```

Ed il ruolo Ansible (assumendo a scopo esemplificativo che il software contenuto nel pacchetto Debian necessiti di un file di configurazione):

```
- name: Install archive-server package
  apt:
    name: archive-server
    state: present
- name: Configure archive-server
  template:
    src: archive-server.conf.j2
    dest: /etc/archive-server.conf
- name: Start and enable the archive-server systemd unit
  systemd:
    name: archive-server.service
    state: started
    enabled: yes
```

Nonostante la differente implementazione, questo servizio risulta indistinguibile, dal punto di vista di float, dalla versione mostrata precedentemente.

In sostanza, float offre:

- un algoritmo di *scheduling* per assegnare servizi ai server
- un meccanismo di basso livello per eseguire container sui server
- una struttura coerente ed organizzata di metadati per rappresentare i servizi

In più, al di sopra di questo livello base, float include una serie di servizi "infrastrutturali", costituenti nel complesso il suddetto sistema operativo distribuito: questi sono a loro volta implementati in termini dei meccanismi precedenti (cioè sono "servizi float", configurati da una descrizione YAML e dei ruoli Ansible).

L'ultima parte della configurazione di float è l'*inventory* (usando la terminologia di Ansible), ovvero semplicemente un elenco dei server da usare come input per l'assegnazione dei servizi.

4.3 Sull'opportunità di scrivere cose oppure usare ciò che già esiste

Questa visione dell'architettura del sistema non è particolarmente originale, ma si allinea ad una serie di *best practices* e mutamenti culturali in atto nell'industria informatica in generale: questo non per inseguire le mode del momento, ma perché adottare prassi comuni è una garanzia che un domani continueranno ad esserci strumenti che le supportino (al netto dell'incertezza che accompagna ogni tentativo di predizione tecnologica sul lungo termine).

Eppure, per via dei particolari requisiti della nostra situazione, specialmente quello di essere una struttura il più possibilie *autonoma* ed *indipendente*, e di utilizzare solo software libero, abbiamo presto notato come alcuni dei pezzi necessari ad implementare questa visione non esistessero ancora, o non si presentassero in forme utili al problema che volevamo risolvere.

Una volta ridotto il problema al minimo necessario, ovvero avendo rivisto le nostre scelte in modo da ridurre quanto più possibile i "pezzi mancanti", è stato necessario prendere una decisione difficile: meglio cercare comunque di far funzionare qualche soluzione già esistente, seppure inadatta, e gestire i compromessi che ciò richiede, oppure rassegnarsi ad implementare del software da sole che svolga esattamente la funzione richiesta? Scrivere software purtroppo è molto facile, ma scrivere software che poi deve essere mantenuto in funzione per anni è un problema molto più complesso ed una responsabilità non indifferente (quantomeno verso noi stesse nel futuro), ed è dunque una decisione da non prendersi alla leggera. D'altra parte, adottare software esistente può rappresentare talvolta una quantità di lavoro complessivo ancora maggiore: si tratta non solo di mantenere la necessaria integrazione per farlo funzionare all'interno di un sistema per cui non è stato concepito, ma anche di considerare il carico cognitivo nel tempo derivante dall'avere innumerevoli "casi speciali" che non combaciano esattamente con il modello mentale desiderato dell'architettura. Sul lungo termine, l'effetto di questa frizione cognitiva è di aumentare molto la complessità del sistema e diminuire il numero delle persone in grado di mantenerlo e modificarlo.

Abbiamo dunque scelto, in alcuni casi, di scrivere il software che ci era necessario, ritenendo che questo fosse sostenibile a patto di rispettare alcune condizioni:

- puntare alla *sostituibilità*, sfruttando la struttura modulare per isolare componenti che presumibilmente un domani potranno essere scambiati con parti standard;
- sempre grazie alla modularità, scrivere componenti *semplici*, la cui funzione può essere compresa senza per forza doverne analizzare il codice sorgente.

Questi principi hanno guidato l'implementazione di tutti i componenti software *custom* che sono menzionati in questo testo.

4.4 Source Control e Continuous Integration

Un moderno sistema di *infrastructure-as-code* comprende nel suo operato una notevole quantità di informazioni, autenticazioni, configurazioni e software. Abbiamo sempre avuto bisogno di cose come pacchetti Debian personalizzati, ma con l'introduzione dei container il numero di questi artefatti aumenta significativamente.

Diventa dunque necessario un sistema per generare e gestire in maniera automatica queste istruzioni e loro variazioni: fortunatamente oggidì di questi sistemi, popolarizzati da Github, ne esistono diversi. Noi abbiamo

scelto di usare Gitlab[4], perché ha un modello di "gruppi" che rende possibile gestire in modo unificato un grande numero di differenti progetti. Ma sarebbe andato bene qualsiasi altro di questi sistemi che offrisse un meccanismo di *Continuous Integration*, ovvero la possibilità di eseguire degli script ad ogni modifica al codice.

Utilizziamo queste funzionalità per vari scopi, come rigenerare le immagini dei container ogni qual volta il repository git del progetto associato riceve delle modifiche, oppure eseguire dei test per verificarne la correttezza, generare dei pacchetti Debian da distribuire poi sulla nostra infrastruttura, etc. Altre componenti di questo sistema (Gitlab) ricoprono un ruolo fondamentale: per esempio il *container registry*, usato per distribuire le immagini dei container agli host.

Gitlab è a questo punto un componente centrale della nostra infrastruttura: anche se è possibile generare comunque tutte le immagini dei container a mano, ciò non è molto pratico per via del loro numero. Per limitare i problemi di dipendenze incrociate, Gitlab è gestito separatamente dal resto dell'infrastruttura, così da poterlo considerare come un servizio di "terze parti". Una delle conseguenze meno piacevoli di questa importanza è il fatto che abbiamo scelto di non aprire l'istanza Gitlab all'utilizzo da parte del pubblico, come misura cautelativa, limitando però purtroppo le possibilità di contribuzione.

4.5 Struttura del nostro software

È utile a questo punto aprire una parentesi sulla struttura, organizzazione e provenienza del nostro software, ovvero di ciò che poi viene deployato in produzione.

L'importanza di avere un'idea della provenienza del software, e di conseguenza di avere una *policy* su come sceglierne le fonti, deriva da due considerazioni: anzitutto, serve a capire a chi stiamo delegando l'autorità di eseguire codice sulla nostra infrastruttura. Storicamente questo ruolo è delegato alle distribuzioni Linux, che mantengono, nel bene o nel male, determinati standard di qualità e sicurezza, ma in un contesto pubblico in cui è diffusissima la pratica di ricompilare i container dai sorgenti il problema è ritornato ad essere centrale.

A noi piace il modello di *trust* offerto da progetti come Debian, con cui ci siamo storicamente sempre trovati bene: il rate di aggiornamento, imposto dalla frequenza di rilascio di nuove release, ci consente di rimanere al passo con i cambiamenti, e ci fidiamo di chi sviluppa il codice e di chi mantiene la loro infrastruttura. O almeno, pur essendo consapevoli dei limiti di questa fiducia, ci sembra un ragionevole compromesso tra la praticità e la sicurezza. Con il moltiplicarsi delle fonti però stabilire questo trust diventa via via più difficile, quindi ci interessa mantenerne il numero strettamente limitato.

L'altro aspetto critico è la necessità di essere logisticamente preparati a gestire queste (presumibilmente molteplici) fonti di software in modo organizzato: non è molto pratico, per esempio, avere N modi diversi per importare codice esterno nella nostra infrastruttura, specie ora che strumenti potenti come la continuous integration consentono un approccio strutturato.

[4] https://en.wikipedia.org/wiki/GitLab

Ci sono soltanto due tipi di artefatti che ci interessa produrre, a seconda dello specifico contesto di utilizzo di ciascun software:

- Pacchetti Debian. Per questioni ideologiche e di familiarità, tutti i nostri sistemi, ed i nostri container (cf. il capitolo *Un approccio pragmatico ai container*) sono basati su Debian. Inoltre, l'infrastruttura richiede la presenza su ciascun server di alcuni componenti che debbono necessariamente essere sotto forma di pacchetto Debian.
- Immagini di container, il "mattone" fondamentale utilizzato dall'infrastruttura. Questo secondo artefatto spesso include il primo: la grande maggioranza delle immagini dei container che utilizziamo sono basate su Debian, dunque cerchiamo ove possibile di installarvi il software necessario sotto forma di pacchetti Debian.

4.5.1 Regole comuni per la generazione di artefatti

Dovendo produrre una varietà così limitata di artefatti, ha senso utilizzare delle regole comuni per la loro generazione, adottando una standardizzazione dei processi di *build* della continuous integration. Per fortuna Gitlab CI supporta la possibilità di includere *script* esterni nella configurazione della CI, dunque abbiamo creato due progetti separati contenenti regole per la produzione di artefatti comuni:

- pipelines/debian[5] per costruire pacchetti Debian. Le regole si aspettano una directory *debian/* nella home del repository in questione, e compilano il pacchetto in parallelo per diverse versioni di Debian: di solito *oldstable* e *stable*, ma quando Debian sta per rilasciare una nuova versione stabile cominciamo a compilare pacchetti anche per quella.

  I pacchetti generati sono pubblicati su un repository Debian autonomo, che abbiamo purtroppo dovuto implementare noi (Gitlab non offre ancora la possibilità di gestire un repository Debian): trattasi di una semplice macchina virtuale con un server web, ed un semplice software[6] per gestire upload di pacchetti tramite SSH.

- pipelines/containers[7] per costruire immagini di container. Queste regole sono molto più semplici (in pratica solo "docker build") ma comprendono anche altri stadi utili come una analisi statica delle dipendenze e delle vulnerabilità. La capacità di aggiungere questa *feature* a tutti i build dei nostri container modificando un unico file mostra la praticità di questo approccio.

L'accentramento delle regole di CI consente di gestire passaggi potenzialmente complicati, come il rilascio di una nuova versione di Debian, sistematicamente e con gradualità.

[5] https://git.autistici.org/pipelines/debian
[6] https://git.autistici.org/pipelines/tools/urepo
[7] https://git.autistici.org/pipelines/containers

4.5.2 Gestione delle dipendenze

Un argomento estremamente importante per noi è la *riproducibilità dei build*: vogliamo essere in grado di produrre lo stesso artefatto, a partire dallo stesso codice, in qualsiasi momento, indipendentemente dallo stato di sistemi esterni. Ciò è fondamentale per poter essere noi a decidere il calendario della progressione degli aggiornamenti, e dunque il momento e la velocità con cui si introducono cambiamenti, anziché essere dipendenti dal ritmo imposto dalle release dei vari software. Ora, ci sono delle difficoltà, volendo riconciliare questo desiderio con l'utilizzo di una distribuzione Debian-like, dove convenzionalmente i pacchetti si auto-aggiornano all'interno di una stessa *major release*: dunque abbiamo deciso che per quanto ci riguarda, tutti i pacchetti in una stessa major release Debian sono alla stessa "versione". In altre parole, consideriamo di avere una dipendenza da *buster* o *bullseye*, anziché dalla versione X di uno specifico pacchetto. Questo è accettabile perché Debian è diventata piuttosto brava, negli anni, a non rompere il software con gli aggiornamenti di sicurezza.

Colmare il varco tra la fonte primaria del software e l'artefatto finale comporta affrontare il problema di come gestire gli aggiornamenti upstream, che è una fonte di lavoro potenzialmente interminabile, e va dunque tenuta sotto controllo con procedure quanto più possibile automatiche. Rispetto a questo si possono identificare quattro possibili approcci:

- Software *stock*, ovvero usato così come proviene dall'*upstream* Debian senza modifiche. Costituisce la stragrande maggioranza di tutto il software che usiamo.
- Software che usiamo senza modifiche ma che non è presente in Debian. Qui dobbiamo distinguere diversi scenari, perché sebbene la risposta giusta sarebbe "portare questo software in Debian" la cosa purtroppo è nel suo complesso al di fuori della nostra portata (anche per via dei tempi necessari), dunque bisogna esaminare dei compromessi:
  - Se il software è abbastanza semplice da pacchettizzare, non svolge ruoli critici dal punto di vista della sicurezza, oppure è parte dell'infrastruttura base di float (e dunque non in un container), ci incarichiamo della pacchettizzazione Debian: creiamo un repository git dedicato con un *fork* del progetto originale e una directory *debian/* per poter usare la nostra CI per creare i pacchetti.
  - Se l'upstream fornisce un proprio repository di pacchetti Debian, e se riteniamo che questo repository sia gestito bene, possiamo con riluttanza considerare di usare questi pacchetti. Non tutti i repository di progetti anche popolari sono gestiti bene purtroppo, e in più di una occasione abbiamo dovuto "freezare" dei pacchetti copiandone a mano una specifica versione nel nostro repository Debian.

  Entrambe le soluzioni indicate comportano un moderato carico di lavoro addizionale, dunque cerchiamo di mantenere il numero di questi software il più limitato possibile (il gruppo Gitlab ai3/thirdparty[8]).

[8] https://git.autistici.org/ai3/thirdparty/

- Software stock cui dobbiamo applicare delle piccole modifiche fatte da noi: questo è il caso più oneroso in quanto dobbiamo tenere traccia degli aggiornamenti upstream e compatibilità con i nostri, e dunque cerchiamo di evitarlo quanto più possibile. All'atto pratico anche questo caso è gestito con un *fork* sul nostro Gitlab del progetto originale, cui poi si aggiungono le nostre *patch*. Un esempio macro di questo caso è Noblogs rispetto a Wordpress.
- Software scritto da noi: in questo caso semplicemente generiamo direttamente dei pacchetti Debian, o dei container, a seconda del tipo di artefatto desiderato.

Detto questo, oggi esistono vari strumenti automatizzati che sono in grado di analizzare la struttura delle dipendenze dei progetti, integrandosi con i sistemi di source control come Gitlab. Questi strumenti consentono di gestire, semi-automaticamente e con relativo poco sforzo, una mole notevole di software con dipendenze sia interne che esterne, ed abilitano sul piano pratico il modello di gestione esplicita delle dipendenze "versionate". Il vantaggio di questo approccio non è solo che ogni build è riproducibile, ma anche che, volendolo, i cambiamenti passano attraverso un controllo esplicito.

In particolare gli strumenti che utilizziamo sono:

- Per rigenerare le immagini dei container risalendo la gerarchia delle relative dipendenze, abbiamo dovuto scrivere un piccolo tool apposito[9] (Gitlab non offre questa funzionalità in modo nativo). In questo modo possiamo sfruttare senza problemi la gerarchia delle immagini e lasciar aggiornare tutto all'automazione.
- Per gestire le dipendenze esplicite, utilizzate da sistemi di *version pinning* (comuni in Go, Python, Javascript, etc.), usiamo un tool chiamato renovate[10], che controlla periodicamente gli aggiornamenti delle dipendenze su Github ed altre fonti. Questo tool è pensato per integrarsi con il normale workflow di revisione delle modifiche: esso crea delle *merge request* su Gitlab, e si può anche configurarlo per fare un *merge* automatico se i test passano con successo.

4.6 Segreti, meta-configurazione ed *environments*

Il desiderio di poter creare "ambienti di test" a piacere, su cui sperimentare in privato o in gruppo, determina alcune conseguenze molto importanti che vale la pena esaminare.

Anzitutto c'è da chiedersi: cos'è un ambiente di test? È una versione degli stessi servizi di A/I (o un loro sottoinsieme, per convenienza), che gira su un insieme di macchine dedicato differente da quelle di produzione – generalmente delle *virtual machines* create al volo per l'occasione. Questo ambiente di test differisce da quello di produzione fondamentalmente perché, oltre a girare su altri host, usa dei *segreti* ed una configurazione

[9] https://git.autistici.org/ai3/tools/gitlab-deps
[10] https://github.com/renovatebot/renovate

differenti. Chiaramente è vuoto dei dati delle utenti, o ha dei dati test per l'occasione e lo si può avere in locale se si desidera.

I *segreti* in questo caso sono cose come password e chiavi crittografiche, e la necessità di supportare ambienti di test agili ci ha costretto a parametrizzarli tutti: la configurazione dei nostri servizi non contiene direttamente alcun segreto, ma solo riferimenti ad essi, e le istruzioni per generarli.

Analogamente, la configurazione dei servizi espone altri parametri di alto livello relativi all'*identità* e altre specificità: per fare un esempio, la configurazione dei nostri servizi di posta descrive solo come costruire un servizio di posta "come il nostro", ma c'è poi un parametro specifico per dire "e questo servizio di posta si chiama autistici.org".

Vediamo così che sono necessari quattro componenti per definire esattamente un ambiente specifico (che d'ora in poi chiameremo *environment*):

- la configurazione generica dei servizi
- un elenco di host (*inventory* nella terminologia di Ansible)
- un insieme di segreti, che possono essere generati automaticamente se necessario
- un elenco di parametri di configurazione di alto livello, che chiameremo *meta-configurazione* essendo la configurazione-della-configurazione

All'atto pratico dunque, ciò che rende i servizi di "autistici.org" in produzione tali è la combinazione della configurazione generica, degli specifici host di produzione (almeno nella misura in cui hanno gli IP che ha anche il registrar del dominio "autistici.org" per le deleghe DNS), e di un insieme di credenziali crittografiche e variabili (specificanti cose come "il dominio da usare è autistici.org"). Variando questi parametri si possono dunque creare altre installazioni, che assomigliano arbitrariamente a quella di produzione.

Questa molteplicità è riflessa nell'organizzazione dei nostri repository:

- ai3/config[11] contiene la configurazione generica dei servizi.
- ai3/prod contiene meta-configurazione e segreti per l'ambiente di produzione (usiamo Ansible Vault per mantenere i segreti crittati nel repository).
- ai3/testbed contiene uno script che automatizza la creazione di ambienti di test sul proprio computer o su una macchina remota, usando Vagrant.

4.7 Un approccio pragmatico ai container

Avendo deciso di comporre i servizi usando dei container, bisogna sviluppare delle linee guida su come questi vadano costruiti, standardizzando metodologie ed approcci così che tutti i nostri container funzionino all'incirca nello stesso modo.

[11] https://git.autistici.org/ai3/config

La visione più diffusa oggi per la strutturazione di un servizio in uno o più container segue il modello "un container == un processo". Noi invece la vediamo diversamente: per noi è meglio se un container rappresenta un'aggregazione di alto livello (anche se magari non esattamente un servizio *float*), con una precisa e completa interfaccia da e per il resto del mondo. Non ci vediamo niente di male, entro limiti ragionevoli, ad avere più di un processo (più di un demone, diciamo) in un singolo container se questi sono parte integrante dello stesso servizio, e se sono fatti per essere co-locati.

Alcuni esempi pratici possono chiarire meglio la questione:

- ci sembra superfluo usare due container diversi per un demone e per il suo exporter Prometheus - integrarli nello stesso container consente tra le altre cose di usare meccanismi "privati" come socket UNIX per far comunicare i due processi.
- alcuni servizi come *lurker* ed *helpdesk* comprendono dei server SMTP che sono parte integrante della API del servizio, e son fatti per parlare con il processo principale esclusivamente in locale: anche qui ha senso eseguire il processo *smtpd* dentro lo stesso container.
- in altri casi i servizi hanno componenti che seguono chiare linee di demarcazione interne, e che dunque ha senso suddividere in più container, per esempio *pannello* e *memcache*, *irc* ed i suoi services, *zipkin* ha un container per l'applicazione web ed uno per il batch processing, etc.

Avendo più processi dentro uno stesso container è necessario un demone *init* dentro il container. Però si pone un problema di gestione del *lifecycle*: cosa fare quando un processo muore, magari per un errore? Qui bisogna fare delle considerazioni che sono legate alla specifica implementazione di *float*.

Per via del fatto che la "superficie di controllo" dell'automazione è al livello della unit systemd (e che lì per esempio si trova il monitoraggio degli errori, ed altro), a noi conviene che gli errori fatali dei servizi emergano a tale livello: quindi il container deve terminare quando c'è un problema fatale con i processi.

Esiste poi una seconda categoria di processi che ci fa comodo trattare in maniera più rilassata: per esempio un errore in un exporter Prometheus non è particolarmente interessante, il processo può essere riavviato dentro il container stesso, senza disturbare il processo principale.

Un ultimo requisito è la possibilità di lanciare uno script prima dell'avvio dei processi principali: così infatti normalmente gestiamo cose come aggiornamenti dei database etc.

Per ora la lista di demoni di init con le features descritte è sfortunatamente breve:

- Chaperone[12], è la soluzione che avevamo scelto di usare al principio. Funzionava bene ma aveva lo svantaggio di essere scritta in Python 3, che vuol dire che ci tiravamo dentro tutto l'environment Python 3 in ogni immagine. In ogni caso il progetto è stato abbandonato qualche anno fa.
- s6-overlay[13] è la soluzione attuale, basata su s6, un init system minimale.

[12] https://chaperone.readthedocs.io/
[13] https://github.com/just-containers/s6-overlay

4.7.1 Relazioni tra immagini

Per ridurre le risorse necessarie a distribuire e far funzionare questi container, conviene organizzare i servizi in una sequenza di immagini via via sempre più specializzate, a seconda della loro generalità. Per esempio, i nostri servizi che usano Apache hanno il seguente annidamento di immagini:

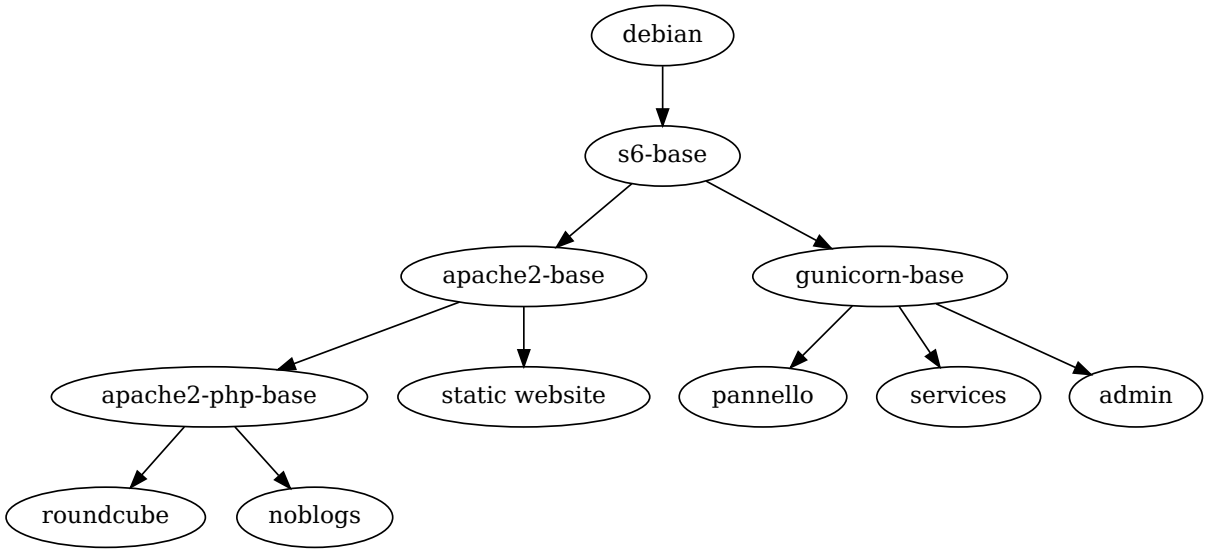

Figura 4.1: Gerarchia parziale delle immagini di container dei nostri servizi basati su Apache

In particolare, attualmente abbiamo a disposizione le seguenti immagini, tutte basate su una versione minimale di Debian stable (*debian:stable-slim*):

- s6-base[14], Debian base più un *supervisor* minimale (s6-overlay[15]), che serve per far girare più di un processo nel container.
- apache2-base[16], un'immagine basata sulla precedente che offre Apache2, configurabile a piacere dalle immagini "figlie". Esempio: roundcube[17].
- gunicorn-base[18], un'immagine con *gunicorn*, per eseguire applicazioni web scritte in Python.

Questa modularità ci assicura che servizi simili abbiano un'implementazione comune, e che, per fare un esempio importante, esista un unico luogo dove modificare la configurazione Apache di tutti i nostri servizi che usano Apache (impedendo dunque di "dimenticarsi" di aggiornarne qualcuno). Inoltre, per via dell'implementazione a *layer* delle immagini dei container, l'utilizzo di questa gerarchia di immagini riduce

[14] https://git.autistici.org/ai3/docker/s6-base
[15] https://github.com/just-containers/s6-overlay
[16] https://git.autistici.org/ai3/docker/apache2-base
[17] https://git.autistici.org/ai3/docker/roundcube
[18] https://git.autistici.org/ai3/docker/gunicorn-base

drasticamente la quantità di dati che devono essere trasferiti su ciascun server, cosa molto importante per esempio in un contesto di test.

4.7.2 Configurazione

Per molti servizi, la configurazione necessaria può essere passata nell'environment e dunque specificata in *services.yml*: in questo caso basta configurare il servizio nell'immagine in modo da usare queste variabili di environment.

Alcuni servizi hanno però bisogno di una configurazione più complessa di quel che si può specificare nell'environment: in questi casi la configurazione verrà creata in Ansible, in una directory dedicata sotto */etc*, la quale sarà poi montata nel container a *runtime* (aggiungendo la directory ai mountpoint in *services.yml*).

Spesso conviene modificare leggermente l'applicazione (o usare uno script di avvio per prepararla) di modo da poter usare un'unica directory per tutta la configurazione necessaria, così da semplificare l'interazione tra Ansible ed il container. Un esempio di questa strategia per un'applicazione PHP, con un layout di configurazione complesso, è in roundcube[19]: qui piazziamo dei link simbolici dentro l'immagine ad una directory */etc/roundcube* che nell'immagine non esiste, ma verrà montata a runtime.

4.7.3 Utenti e permessi

Normalmente *float* crea un utente ed un gruppo dedicati per ciascun servizio, ed esegue i container con quell'utente *all'interno del container*. Questo implica che non è possibile sapere quale sia questa utente al momento della costruzione dell'immagine Docker: è a tutti gli effetti un parametro applicato a runtime. Bisogna quindi avere cura di costruire immagini Docker che possano essere eseguite da un utente qualsiasi. All'atto pratico questo significa:

- configurare il servizio in modo da non fare setuid()
- se il servizio deve scrivere dati su disco:
  - per dati effimeri, configurare il servizio per usare */tmp* oppure */run/lock*, a seconda, che siano automaticamente montati da *float* come tmpfs con i permessi giusti
  - per dati che devono persistere attraverso i restart del container, bisogna creare (in Ansible!) una directory con i permessi giusti, e montarla nel container
- i dati delle utenti relativi al servizio dovranno essere leggibili (e scrivibili, a seconda dell'applicazione) dall'utente dedicata, e montati anch'essi in una directory opportuna (per es. */home/mail*).

L'utilizzo di porte riservate (< 1024) non rappresenta un problema in quanto possiamo impostare esternamente delle *capabilities* come `CAP_NET_BIND_SERVICE`.

[19] https://git.autistici.org/ai3/docker/roundcube

Questo sistema ci consente in pratica un ulteriore livello di isolamento dei permessi per garantire che un servizio non abbia accesso in scrittura a dati sensibili come la propria configurazione, o all'immagine stessa del container, in quanto limitato dai permessi UNIX senza necessariamente dipendere dall'aver montato l'immagine del container come *read-only* (cosa che comunque viene fatta manipolando il mountpoint *root* nel namespace del container).

Ripensandoci, abbiamo dovuto fare un lavoro non nullo per garantire che ciascuno dei nostri container potesse venire eseguito come un utente arbitraria *all'interno del container*. Un'alternativa avrebbe potuto essere definire un utente standard, per esempio UID 1000 (per non aver bisogno di un ulteriore campo di metadati nella descrizione del servizio), configurare i container per usare quello entro il container con la direttiva `USER`, e dire a float di mappare l'UID interno con l'utente *esterna*.

4.7.4 Porte

Il layout della rete nella nostra infrastruttura è volutamente molto semplice: anziché adottare una strategia complessa di isolamento dei container a livello di rete, come fanno soluzioni avanzate come Kubernetes, abbiamo deciso di mantenere un modello concettualmente più affine all'amministrazione di sistema tradizionale. I container condividono il namespace di rete con l'host, dunque devono necessariamente utilizzare ciascuno delle porte differenti. Ci affidiamo poi al firewall di sistema per assicurarsi che queste porte siano raggiungibili soltanto attraverso la VPN interna e non dall'esterno.

Tra le conseguenze di questa scelta c'è anche il fatto che non è possibile allocare più di una istanza dello stesso servizio su di un host (ci sarebbe un conflitto di porte). Un'altra conseguenza ha a che fare con il modo in cui il container si aspetta di venire eseguito: normalmente le immagini Docker esportano le porte standard dei servizi che implementano, che poi vengono rimappate a runtime a seconda delle esigenze. Con il nostro approccio invece ci aspettiamo che il container bindi ad una porta che possiamo specificare a runtime attraverso una variabile di environment, così che per Docker il mapping delle porte sia sempre 1:1.

Per fare ciò, l'immagine non deve hard-codare una specifica porta per il servizio, ma usare una variabile di environment (di preferenza, in quanto più semplice da integrare nella configurazione dei servizi) o un altro meccanismo di configurazione. Questo rende alcune immagini preconfezionate non direttamente usabili sulla nostra infrastruttura (poco male, visto che comunque cerchiamo di non usarne), e ha richiesto qualche accorgimento.

Un paio di esempi differenti:

- l'immagine apache2-base[20] e tutti i suoi derivati usano la variabile `APACHE_PORT` per selezionare la porta da usare (apache2 cortesemente può utilizzare variabili di environment direttamente nei suoi file di configurazione). L'accortezza da notare è che bisogna usare `${APACHE_PORT}` nelle direttive `<VirtualHost>` dei siti configurati:

[20] https://git.autistici.org/ai3/docker/apache2-base

```
<VirtualHost *:${APACHE_PORT}>
    ...
</VirtualHost>
```

- l'immagine pannello[21], che usa gunicorn-base, legge la variabile di environment `BIND_ADDR` direttamente dal file di configurazione gunicorn.conf[22].

4.8 Comunicazione interna e RPC

Una architettura come quella che descriviamo in questo documento comporta una elevata quantità di comunicazione interna tra servizi, sotto forma di RPC (*remote procedure call*), distinta dal flusso dati direttamente generato dagli utenti con protocolli specifici come può essere per le richieste HTTP provenienti dall'esterno o per il traffico di posta SMTP o IMAP.

Ci sono molte possibilità per affrontare questo problema tecnico, motivo per cui abbiamo deciso di standardizzare le chiamate RPC interne su una soluzione unica, almeno per quanto riguarda il software scritto da noi, dove esiste la possibilità di scelta. Pur conoscendo possibilità elaborate come GRPC[23], abbiamo invece deciso per un protocollo semplice e banale: richieste HTTP con contenuto e risposte JSON. Questo non è un protocollo particolarmente furbo né tantomeno performante, ma il livello di traffico interno è comunque relativamente basso, ed abbiamo preferito una soluzione che fosse semplice, facilmente comprensibile e diagnosticabile con strumenti che tutti conoscono.

Per questo motivo abbiamo scritto due librerie, in Go[24] ed in Python[25], i due linguaggi che al momento usiamo per scrivere software. Queste librerie implementano il minimo di funzionalità che riteniamo necessarie per un layer RPC efficace in questo contesto:

- un qualche meccanismo di *load balancing* per distribuire le richieste tra diversi backend identici, nel nostro caso un algoritmo *round-robin* è più che sufficiente;
- un sistema di *back-off* con intervalli esponenziali per ritentare le richieste più volte in caso di errori temporanei, di rete o altro;
- un modo per assicurarsi che ogni richiesta RPC abbia un timeout associato, per evitare che le richieste possano rimanere "appese";
- supporto per la mutua autenticazione dei servizi con mTLS;
- lato server, un meccanismo di protezione consistente in un limite al numero massimo di richieste concorrenti, dopo il quale il server ritorna un errore temporaneo di sovraccarico.

[21] https://git.autistici.org/ai3/pannello
[22] https://git.autistici.org/ai3/pannello/blob/master/docker/gunicorn.conf
[23] https://grpc.io
[24] https://git.autistici.org/ai3/go-common
[25] https://git.autistici.org/ai3/tools/python-web-common

L'insieme di queste caratteristiche, pure molto semplici, consente comunque ad un servizio di comunicare con un altro con successo anche in presenza di istanze con errori o irraggiungibili: in questi casi, si noterà un incremento della latenza di una frazione delle richieste. Scegliendo opportunamente i timeout si può decidere in quale misura questo rappresenta un inconveniente per l'utente finale. È un sistema migliorabile lungo molte dimensioni: l'aggiunta di *circuit breakers* risolverebbe il problema della latenza in presenza di errori, mentre algoritmi migliori di load balancing potrebbero tenere conto del carico dei server nella distribuzione delle richieste, ed un fallback preferenziale per i backend locali potrebbe ridurre significativamente le latenze medie. Già l'adozione di questi principi ci offre però un servizio di qualità sufficiente, ovvero in grado di trarre effettivo vantaggio dagli elementi replicati della nostra architettura.

L'isolamento della funzionalità RPC in delle librerie ci consente comunque di cambiare tecnologia con il minimo sforzo possibile se dovessimo decidere così in futuro.

5 Struttura dei dati e dei servizi

5.1 Database delle utenze

Come si può dedurre dal nome, il *database delle utenze* contiene tutti i dati relativi ai vari account delle utenti sui servizi che offriamo. Questa è una definizione piuttosto vaga, che si può precisare chiarendo che questo database contiene oggetti di due tipi:

- Gli *account* veri e propri, che corrispondono ad una specifica pseudo-identità, e cui sono associati i dati relativi all'autenticazione. Un account è identificato da un indirizzo email.
- Varie *risorse*, rappresentanti specifiche istanze dei vari servizi che offriamo (una casella email, uno spazio web, un database, etc).

Risorse ed account sono organizzati in modo gerarchico e nella maggioranza dei casi, ogni specifico account "possiede" delle risorse in via esclusiva. Esistono però anche risorse "primarie", come le mailing list, che non hanno necessariamente una attribuzione esclusiva: in questo caso manteniamo traccia dell'associazione solo dove essa esista, per poter mostrare "le tue mailing list" nel pannello utente, ma consentendo anche la gestione di mailing list da parte di indirizzi esterni.

Il database delle utenze contiene attualmente risorse di questi tipi:

- *account email*, caselle di posta (e account XMPP, con cui esiste una corrispondenza implicita 1:1);
- *mailing list*;
- *newsletter*, che sono un tipo differente di mailing list configurate per la comunicazione unidirezionale, e gestite da un'infrastruttura separata;
- *account DAV*, dal nome del protocollo utilizzato per accedervi: uno spazio disco destinato a servire siti web;
- *siti web*, suddivisi in *domini* e *sottositi* (sottodirectory di autistici.org e inventati.org);
- *database MySQL* usati per il servizio di web hosting.

Alcune risorse (DAV, MySQL) possono a loro volta contenere ulteriori dati di autenticazione specifici del servizio cui si riferiscono.

La relazione tra account DAV e siti web è sfortunatamente non gerarchica, in virtù di una situazione storica in cui c'è stata incertezza sulle modalità esatte con cui organizzare questa relazione: uno-a-uno (un

sito per account DAV e viceversa), uno-a-molti (più siti per un account DAV), o molti-a-molti (libera associazione di account DAV, per esempio limitati a specifiche sottodirectory, e siti). Storicamente ci siamo ritrovati nell'ultimo scenario, ma non abbiamo purtroppo inserito un altro oggetto nella gerarchia per poter raggruppare correttamente le cose: il risultato è che questo raggruppamento viene fatto all'atto della presentazione, nel pannello utente, andando a guardare gli specifici *path* delle varie risorse.

Un modello dati di questo tipo, per giunta implementato su un database non relazionale, richiede di mantenere manualmente un gran numero di invarianti, per esempio:

- dato il disaccoppiamento tra *identità* e *risorse*, è necessario che esista una risorsa *email* per ciascun account, il cui indirizzo email corrisponda al nome dell'account;
- alcuni servizi necessitano di risorse correlate (uno spazio web, ed i siti ad esso associati per esempio), nel qual caso è necessario verificare che siano sullo stesso server;
- ogni sito web deve avere un account DAV associato affinché esista una directory sul filesystem da cui servire;
- alcune risorse correlate hanno attributi *denormalizzati* (riprodotti identici tra più risorse correlate), cosa che semplifica enormemente l'automazione, ma rende necessario assicurarsi che questi rimangano coerenti.

Per fortuna tutti i servizi usano questo database in modalità *read-only* quindi non è lì che ci si deve preoccupare di mantenere queste invarianti. Questa complessità è contenuta ed isolata all'interno di un unico *componente di amministrazione* (accountserver[1]), con una API ben definita, che è l'unico servizio autorizzato a creare e modificare oggetti nel database. Aver potuto concentrare tutta la complessità e la non-ovvietà derivata da scelte passate magari non più ottimali (LDAP, il particolare schema utilizzato) in un unico software è stato il punto di partenza per il rinnovamento dei sistemi.

5.2 Dati e servizi partizionati

La maggior parte dei nostri servizi adotta una strategia di *scaling* detta *partizionamento* (o *sharding*), in cui, in presenza di più di un server, i dati di ciascun account sono associati ad uno specifico server e solo a quello. Questo in alternativa alla *replicazione*, situazione in cui i dati di un account sono riprodotti su più server.

La differenza fondamentale tra i due approcci, almeno dal punto di vista che ci interessa, sta nel loro comportamento a fronte del fallimento di un server: la replicazione consente di continuare ad offrire il servizio, mentre nel caso del partizionamento, le utenti i cui dati si trovano sul server fallito riscontreranno problemi di accesso. Se dunque la replicazione è nettamente superiore, perché non sceglierla? Ci sono diversi motivi:

[1] https://git.autistici.org/ai3/accountserver

- un *divario tecnologico*: all'atto della progettazione di questa architettura, non esisteva un prodotto software in grado di fornire uno storage POSIX-compliant globale affidabile e performante, almeno non al livello di impegno e competenza di cui disponevamo all'epoca.
- l'architettura geo-distribuita: per i sistemi di storage POSIX-compliant, la non-località di un deployment globale rappresenta il *worst-case scenario*, per via delle elevate latenze necessarie a mantenere un consenso distribuito in questa situazione. Questo rappresenta un problema per un'applicazione *storage-heavy* come la posta, e suggerisce di sfruttare invece la peculiarità che in questi servizi gli accessi ai dati sono *locali* alle utenze (cioè ciascun utente accede solo ai propri dati).

Nel corso del tempo sono emersi sistemi di storage distribuito estremamente validi che aggirano i problemi menzionati rinunciando alla compatibilità POSIX: ovvero questi sono sistemi cui non si accede "montando un filesystem remoto", bensì utilizzando particolari API di alto livello. Purtroppo questa non è un'opzione applicabile ai nostri servizi: sia nel caso della posta che del web infatti utilizziamo software scritto da terzi (Dovecot, Apache) che non può usare queste API e necessita assolutamente di un filesystem "vero" per funzionare.

La particolare risposta dei sistemi partizionati ai problemi (vero per i sistemi distribuiti in generale) comporta degli adattamenti cognitivi non necessariamente ovvi, sia per chi amministra che per le utenti. Non esiste più infatti un concetto binario di operatività ("il servizio è su" e "il servizio è giù"), questa si esprime invece in funzione dell'esperienza diretta dell'utente: al verificarsi di un problema, il servizio può continuare a funzionare benissimo per alcune utenti, e non funzionare affatto per altre. È dunque necessario modificare i sistemi di monitoraggio e diagnostica per esprimersi in maniera quantitativa, non più "problema sì" / "problema no" ma "una percentuale X di utenti sta avendo problemi". Ciò implica chiaramente un aumento del carico cognitivo necessario a diagnosticare i problemi, che è mitigabile con opportuni strumenti analitici: nel nostro caso, questo è stato ottenuto investendo tempo nella funzionalità del sistema di monitoraggio.

5.3 Implementazione LDAP

Per ragioni storiche il database delle utenze utilizza un database LDAP. Questa probabilmente non sarebbe la scelta che faremmo oggi cominciando da zero: LDAP è una tecnologia arcaica ed idiosincratica, richiede competenze specifiche, e le ragioni che ci portarono a sceglierla nel 2005 non sono più tutte valide oggi.

All'epoca, LDAP aveva importanti vantaggi:

- è molto veloce
- possiede un meccanismo di replicazione robusto e relativamente facile da configurare
- ampio supporto lato client (vero anche per SQL del resto)

Nonostante ciò ci sono aspetti negativi:

- modificare lo schema è complesso, richiede la comprensione dei vari tipi di dato LDAP, dei loro comparatori, e delle relative interazioni, etc.
- le API sono di basso livello, è praticamente sempre necessario un layer intermedio per interfacciarsi al database quando si vogliono fare modifiche
- la complessità della gestione del database tende dunque ad essere spostata dal lato del client
- la mancanza di transazioni rende necessario un approccio cautelativo quando si modificano dati: all'atto pratico questo si traduce in un utilizzo "light" del database (preferendo database esterni per dati altamente mutabili) e nell'aver stabilito una stretta policy di *data ownership* per cui c'è una associazione univoca tra attributi LDAP e processi di automazione che li modificano (si veda il capitolo *Automazione*)

Nel corso del tempo abbiamo sempre più isolato gli accessi diretti al database LDAP, consentendoci anche di implementare delle ACL adeguate, fino alla situazione presente:

- tutti gli accessi al database relativi all'autenticazione sono stati accentrati nel server di autenticazione *auth-server* (cf. capitolo *Server di autenticazione*);
- tutti gli accessi amministrativi in scrittura, per creare o modificare oggetti nel database, sono stati accentrati nel servizio *accountserver*, che espone una API di alto livello;
- alcuni servizi hanno accesso diretto a LDAP in sola lettura, per query come verificare l'esistenza di un account, o determinarne parametri specifici per il servizio (per es. il path della mailbox);
- i meccanismi di automazione hanno accesso in scrittura limitato a quegli attributi necessari ad implementare la relativa *state machine* (cf. capitolo *Automazione*).

In termini di architettura del servizio, ci affidiamo alla replicazione LDAP per distribuire server LDAP dovunque sia necessario fare molte query: nello specifico sui frontend e sui server di posta. Il costo della replicazione è molto basso (i dati cambiano di rado), e il database è sufficientemente piccolo. In particolare, è abbastanza piccolo da entrare tutto in memoria: le repliche servono il database da un *tmpfs*, scaricandone una copia nuova (in meno di 30 secondi) ad ogni reboot.

6 Identità ed autenticazione

L'indirizzo email oggi svolge ancora per molti versi il ruolo di *identità primaria* di una persona su Internet, e considerato il nostro ruolo di fornitori di email, abbiamo sentito l'esigenza di aggiornare la nostra infrastruttura per proteggere meglio queste identità.

In particolare sono due le features che ci interessava implementare, per avere una soluzione al passo coi tempi:

- *Two-Factor Authentication* con hardware token, che pur comportando un costo per l'utente (il suddetto token), è l'unico modo serio per prevenire il *phishing*.
- La possibilità per le utenti di creare a piacere password limitate e depotenziate: *limitate* perché danno accesso ad un singolo servizio, e *depotenziate* in quanto non consentono l'accesso ad operazioni privilegiate (come il cambio della password), prevenendo il *takeover* dell'account. Questo consente alle persone di gestire il loro profilo di rischio a livello di singolo dispositivo.

Naturalmente avendo utenti pre-esistenti è stato necessario anche continuare a gestire il consueto workflow basato su semplici username e password.

Infine un sistema moderno di gestione di identità deve anche essere un minimo robusto rispetto al *brute-forcing* ed altri tentativi di abuso automatizzati che oggi sono molto diffusi.

In questo scenario, la logica di validazione delle credenziali diventa piuttosto complicata (si pensi a criteri necessari come "permettere il login con la password primaria solo se non sono definite password specifiche per questo servizio"), per cui ha senso mantenerla in un unico luogo anziché distribuirla tra le varie applicazioni. Questa ed altre considerazioni ci hanno spinto verso l'implementazione di un servizio di autenticazione autonomo centralizzato, a cui i vari servizi delegano l'autenticazione delle utenti.

Anche in questo caso i sistemi di autenticazione free ed open source non avevano all'epoca le funzionalità che ci erano necessarie, ed abbiam deciso di scrivere un software ad hoc. La situazione è in costante miglioramento e molte di queste pratiche stanno diventando comuni, quindi anche questa come molte altre parti della nostra infrastruttura potranno essere sostituite con componenti *stock* a tempo debito.

Esaminiamo dunque in dettaglio i vari modi in cui si articola l'autenticazione delle utenti e come funzionano le soluzioni che abbiamo implementato.

6.1 Server di autenticazione

Al livello più basso dello stack di autenticazione sta il server principale di autenticazione, o auth-server[1]. Lo scopo di questo servizio è soltanto quello di autenticare le utenti, utilizzando i dati memorizzati nel database. La decisione di autenticare un utente o meno viene fatta in base alle credenziali presentate (password, 2FA), ed eventualmente ad altre informazioni contestuali alla richiesta (come l'indirizzo IP).

È importante notare che parliamo di *servizio centralizzato*, ma in realtà il servizio è implementato usando istanze multiple dell'auth-server, evitando di costituire un *single point of failure*.

Il server di autenticazione espone un protocollo testuale molto semplice, che però supporta tutte le informazioni aggiuntive (oltre ad username e password) necessarie per la two-factor authentication. Anche qui purtroppo non abbiamo trovato un protocollo esistente che supportasse il nostro caso e mantenesse bassi i costi di implementazione (abbiamo pensato che implementare correttamente un server RADIUS fosse fuori dalla nostra portata).

Ci sono due tipi di client differenti:

- I client sofisticati, in grado di gestire un workflow 2FA con l'utente, che possono fare uso completo del protocollo di autenticazione. All'atto pratico, l'unico modo di implementare un workflow 2FA è con HTTP, e tutta l'autenticazione HTTP è gestita con *Single Sign-On* (vedi capitolo successivo), dunque esiste un solo client di questo tipo ed è il servizio di Single Sign-On.
- I client che, per via di limiti nel protocollo che il servizio associato parla con le utenti, implementano l'autenticazione esclusivamente in termini di username e password. Questo include in pratica tutti i servizi non-HTTP.

Per integrare il servizio di autenticazione con questa seconda tipologia di client è stato necessario scrivere della "colla" sotto forma di un modulo PAM, lo standard UNIX per l'autenticazione dei servizi. Va detto che l'esistenza di questo modulo PAM, necessariamente scritto in C, è la ragione principale dietro la scelta di un protocollo *custom*: è molto più facile implementare in C un protocollo testuale line-oriented che un client HTTP/JSON. Come con molte altre scelte, dovessimo cambiare idea in proposito la compartimentazione dei servizi ci consentirà di sostituire questa implementazione con poco sforzo.

Il servizio di autenticazione è stato scritto per essere il più generico possibile (supporta per esempio anche backend SQL), ed una delle caratteristiche interessanti è il supporto supporto per multipli database di utenti. Usiamo questa feature per separare completamente le admin (componenti di A/I) dalle utenti "normali": mentre i secondi sono memorizzati nel nostro database LDAP, le utenze amministrative sono memorizzate in un file statico, distribuito mediante configuration management. Questo ci permette di continuare ad usare gli strumenti amministrativi via web anche se il database LDAP dovesse avere problemi.

[1] https://git.autistici.org/id/auth

6.2 Single Sign-On

Single Sign-On è un termine un po' vago che oggi viene associato ad uno spettro di tecnologie che vanno dalla federazione delle identità alla delega di accesso ai dati. A noi interessa l'accezione in cui lo si intende come l'unificazione del controllo dell'identità attraverso varie applicazioni distinte, fornite da un'unica organizzazione. Il dominio è ben definito: le utenti sono le utenti di autistici.org, le applicazioni sono quelle offerte da autistici.org alle proprie utenti. Nello specifico, ci interessano le applicazioni *web*: non esiste praticamente supporto per *SSO* nei client per altri protocolli, mentre il web è sufficientemente flessibile nella sua struttura da consentire la costruzione di workflow di autenticazione arbitrari.

L'esigenza di questo tipo di tecnologia viene da due requisiti distinti ma convergenti:

- Il desiderio di unificare, dal punto di vista dell'esperienza dell'utente, due applicazioni web separate: il pannello di gestione dell'account, e la webmail. Siccome usiamo il pannello di gestione come strumento di comunicazione primaria con le utenti (nonché per implementare funzionalità come l'obbligo di cambio di password), vogliamo enfatizzarlo come "punto di accesso" principale ai nostri servizi, e chiedere di autenticarsi una seconda volta per la webmail è un ostacolo in questo senso.
- La necessità di proteggere tutte le varie interfacce amministrative web presenti nell'infrastruttura dietro un unico controllo di accesso per le admin di A/I. Il numero di queste interfacce scoraggia dall'implementare l'autenticazione separatamente per ciascuna di esse.

Da un servizio Single Sign-On moderno, incentrato sul web, ci aspettiamo le seguenti caratteristiche:

- Deve basarsi su *token* firmati, che attestano un'identità vincolata ad uno specifico *servizio* (nel nostro caso identificato da una URL), e devono avere una validità strettamente limitata nel tempo. Questi token sono erogati da un unico *servizio di login*, che implementa l'interfaccia per l'autenticazione delle utenti con password e 2FA.
- Deve essere possibile verificare la validità di questi token soltanto possedendo la chiave pubblica del servizio di login, senza nessun bisogno di coordinarsi con un servizio di autenticazione centralizzato.
- Nel caso in cui servizi diversi siano in comunicazione internamente, deve esistere un meccanismo per delegare l'autorità dell'utente autenticato. Per esempio, dev'essere possibile per un servizio A ottenere, presentando un token valido per un utente, un altro token per un servizio B, valido per lo stesso utente, per poter così trasferire l'identità verificata dell'utente attraverso lo stack dei servizi. L'elenco delle possibili transizioni A -> B dev'essere rigidamente controllato.
- Offrire tutte le integrazioni di cui abbiamo bisogno:
  - un modulo di autenticazione per Apache, il web server che utilizziamo per applicazioni come la webmail
  - binding e *middleware* nei vari linguaggi che utilizziamo (Python, Go), per l'integrazione con le nostre applicazioni web che utilizzano un proprio stack HTTP

  - un modulo PAM per i servizi non HTTP
  - un meccanismo per integrarsi con applicazioni più complesse che supportano meccanismi di autenticazione avanzati (SAML, OAuth)

Ciò che poi rende un'implementazione di un tale sistema "incentrata sul web", al contrario di sistemi simili ma di basso livello come Kerberos[2], è per l'appunto il focus sul browser come client primario: in pratica, la possibilità di trasferire i suddetti token assieme alle richieste HTTP (mediante cookies, per esempio), e di controllare il flusso del workflow interattivo usando redirect HTTP.

Oggi esistono diverse soluzioni in ambito free e open source che offrono queste funzionalità, ma quando prendemmo questa decisione il Single Sign-On era ancora una feature *enterprise*, limitata quasi esclusivamente ad aziende e al mondo del business, dunque le opzioni a nostra disposizione erano limitate e poco funzionali. Dopo lunghe riflessioni, e avendo considerato il problema a fondo, abbiamo preso la decisione di provare a scrivere noi stesse una implementazione minimale, ma funzionale, di un servizio SSO.

Il risultato è ai/sso[3], che implementa un servizio di Single Sign-On molto semplice ma con tutte le caratteristiche necessarie, costituito da pochi componenti *stateless*, che si integrano facilmente nella nostra infrastruttura. A distanza di anni possiamo dire che a grandi linee la strategia si è rivelata corretta: adesso cominciano ad esistere alternative *open* compatibili con i nostri requisiti, e come per molte altre soluzioni presentate in questo documento, l'architettura modulare ci consentirebbe di sostituire *ai/sso* con un'altra implementazione (OAuth/OIDC, SAML, etc) con uno sforzo limitato.

È in ogni caso importante tenere a mente che i sistemi SSO basati su cookies, pur avendo molti vantaggi in termini di implementazione, hanno anche qualche limitazione:

- Come tutti i meccanismi basati su *bearer token*, sono soggetti al rischio di furto di identità tramite esfiltrazione del token: essendo parte della richiesta, il token può finire in molti luoghi non facilmente controllabili (lo storage locale del browser, log HTTP, cache, etc). Un modo per limitare questo rischio è tenere bassa la validità temporale del token.
- Il *logout* è un'operazione difficile da definire e da implementare, in quanto si basa sulla coordinazione di varie richieste HTTP per ottenere la rimozione di cookies su vari domini differenti. Non esiste un meccanismo per forzare il logout di una sessione, se non attenderne la scadenza.
- Dato che il meccanismo di autenticazione interferisce, inserendo dei redirect, con le richieste HTTP, questo tende a non funzionare del tutto in caso di richieste POST, rendendo le applicazioni complesse apparentemente più "fragili" di quel che siano. Per fortuna il comportamento istintivo delle utenti in questi casi, ricaricare la pagina, risolve il problema.

Infine bisogna notare che, anche se il client primario per il sistema di Single Sign-On è il browser, ci sono in realtà situazioni in cui vogliamo che altri servizi, non basati su HTTP, supportino SSO. Nel nostro particolare

[2] https://en.wikipedia.org/wiki/Kerberos_(protocol)
[3] https://git.autistici.org/ai/sso

insieme di servizi, dobbiamo anche considerare il caso in cui un servizio web chiami internamente un servizio "non-web": il caso principale è la webmail, in cui l'applicazione web deve parlare con il servizio di posta via IMAP. Nel contesto di Single Sign-On però l'applicazione web non conosce la password dell'utente: è dunque necessario che il server IMAP possa a sua volta accettare e verificare token firmati al posto delle password.

6.3 Crittografia

Un'altra cosa che volevamo assolutamente implementare era la possibilità di crittografare i contenuti della casella di posta di ciascun account in modo che solo l'utente, a conoscenza della password, potesse leggerne i contenuti, e che in mancanza della password ciò non fosse possibile neanche per le admin stessi. Questo è un importante punto di profilo legale a fronte di vari Terabyte di caselle di posta.

Un modo per ottenere questo risultato è usare una chiave crittografica derivata dalla password. Sfortunatamente questo semplice meccanismo ha dei problemi nel momento in cui ci possono essere più di una password, o anche semplicemente se si desidera cambiarle. Quel che si può allora fare è creare una chiave crittografica a lungo termine, usata per crittografare i contenuti dell'utente, e poi usare la chiave derivata dalla password per memorizzarne nel database una copia crittata. Si avranno dunque eventuali molteplici copie della chiave, ciascuna crittografata con una diversa password; inoltre al cambio di password si può memorizzare solo una nuova copia della chiave (crittata con la nuova password) e non c'è bisogno di modificare i dati in alcun modo.

Cosa succede però nel caso di un'applicazione web che usi SSO? In questo caso l'applicazione non ha accesso alla password, eppure bisogna fare in modo che possa avere comunque accesso alla chiave crittografica. La gamma di possibili soluzioni è piuttosto vasta, noi abbiamo scelto così:

- Abbiamo creato un nuovo servizio keystore[4], che opera come una cache di breve termine per le chiavi crittografiche (decrittate in memoria!) delle utenti;
- il servizio di Single Sign-On fa una chiamata a questo servizio all'atto del login, dunque avendo una password valida;
- keystore decritta la chiave dell'utente usando la password e la tiene nella sua cache;
- l'applicazione dietro SSO può fare una chiamata a keystore, presentando delle credenziali valide per l'utente (il ticket SSO), e ricevere la chiave.

In questo modo (e impostando il *time to live* della cache in modo da essere uguale alla durata della validità del ticket SSO), si evita di mantenere chiavi "in chiaro" in memoria per un periodo molto maggiore di quello dell'effettivo utilizzo dell'applicazione, che comunque, ricordiamo, già deve mantenere una copia della chiave decrittata in memoria per poter decrittare i dati dell'utente.

[4] https://git.autistici.org/id/keystore

6.4 Meccanismi di autenticazione

Come si intuisce da quanto descritto finora, il nostro sistema di autenticazione è pensato per supportare diverse modalità di autenticazione per l'utente

- Semplice password. Avendo già un sistema con migliaia di utenti con soltanto una password, non si poteva evitare di continuare a consentire l'autenticazione senza alcun tipo di secondo fattore.

- OTP[5], ovvero un secondo fattore pseudocasuale derivato da un segreto memorizzato su un dispositivo separato (spesso una applicazione su un cellulare, ma è anche possibile utilizzare un token hardware in questo modo). Questo tipo di autenticazione a secondo fattore rimane vulnerabile al *phishing* ma consente comunque l'utilizzo di 2FA senza costi aggiuntivi (se si possiede per esempio già uno smartphone).

- Token hardware (usando WebAuthN[6]), un secondo fattore di autenticazione veicolato da un dispositivo fisico e non vulnerabile al *phishing*.

Questi meccanismi sono direttamente supportati dal servizio centrale di autenticazione SSO, ma da soli non sono sufficienti a coprire i servizi che offriamo, in particolare quelli non HTTP. Per questi altri servizi, la *best practice* adottata è di poter associare al proprio account password secondarie (*application-specific passwords*), che siano però valide soltanto per uno specifico servizio. Questa limitazione fa sì che, dovessero venire compromesse, sia comunque impossibile utilizzarle per assumere il controllo completo dell'account.

È poi responsabilità del server di autenticazione impedire, per esempio, che si possa autenticarsi con il servizio di posta utilizzando la password primaria se sono presenti password application-specific, e altre simili regole che implementano nel loro complesso "l'esperienza" di autenticazione dell'utente.

[5] https://en.wikipedia.org/wiki/One-time_password
[6] https://en.wikipedia.org/wiki/WebAuthN

6.5 Workflow nel dettaglio

6.5.1 SSO

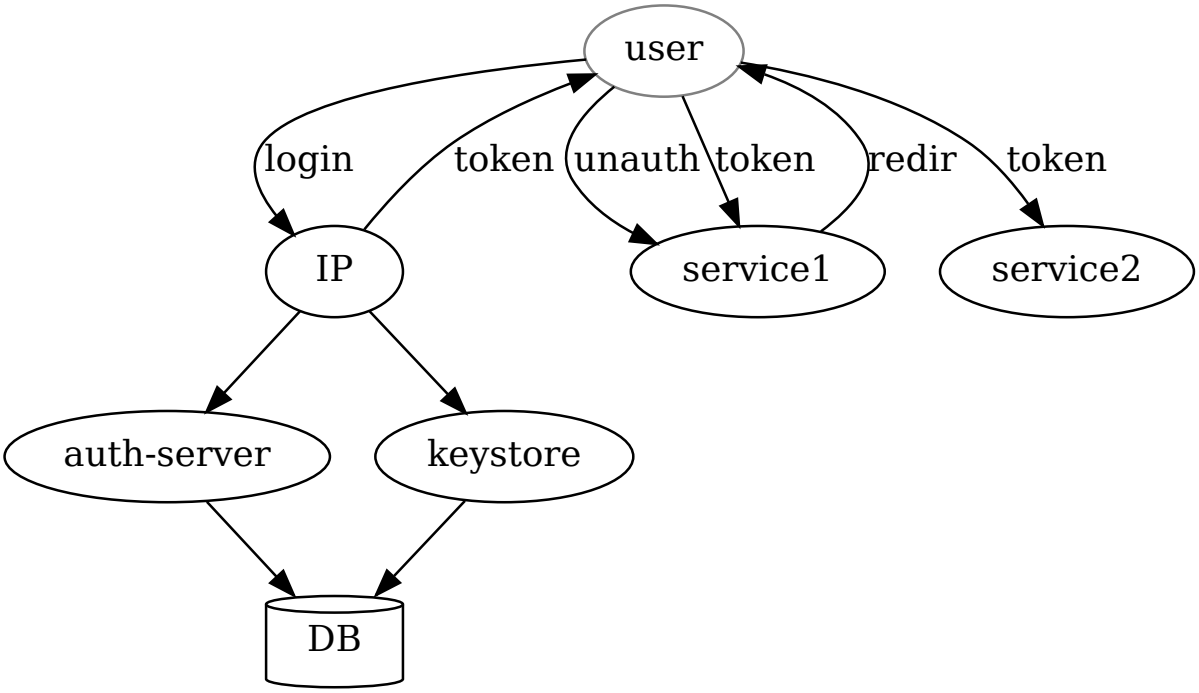

Figura 6.1: Flusso di autenticazione con *Single Sign-On*

Seguiamo in dettaglio una richiesta HTTP fatta da un utente ad un servizio protetto da Single Sign-On:

1. l'utente visita `https://service1/`
2. il server *service1* determina che la richiesta non contiene nessun cookie con un token SSO valido, dunque ritorna un redirect al servizio di login, aggiungendo come parametri sia il nome del servizio stesso (service1) che la URL originariamente richiesta
3. il servizio di login nota che l'utente non ha un cookie di sessione valido, dunque mostra un form di autenticazione
4. l'utente si autentica con username, password, ed un eventuale secondo fattore
5. il servizio di login delega la verifica dell'autenticazione al servizio centralizzato di autenticazione *auth-server* (il quale a sua volta interpella il database delle utenze)
6. se l'autenticazione ha successo, il servizio di login chiama *keystore* per decrittare la chiave di crittazione specifica dell'utente e tenerla nella sua cache in memoria
7. il servizio di login imposta un suo cookie di sessione, così da non dover richiedere autenticazione all'utente una seconda volta, genera un token SSO valido per il servizio *service1*, e ritorna un redirect

al servizio originale *service1*, passando una URL apposita (`/sso_login`) con il nuovo token SSO e la URL originariamente richiesta
8. il server *service1* riceve il token SSO nella URL, lo salva in un cookie, e redirige l'utente alla URL iniziale (sempre su *service1*)
9. *service1* adesso vede un token SSO valido nel cookie, e serve la richiesta normalmente.

Se l'utente nel corso della stessa sessione decide di visitare il servizio *service2*, i passaggi sono più semplici in quanto il servizio di login riconosce il proprio cookie di sessione e può saltare la fase di autenticazione, passando direttamente alla generazione del nuovo token SSO e al conseguente redirect. Dal punto di vista dell'utente questa appare soltanto come una serie di redirect ed è quindi "invisibile". Lo stesso accade anche se l'utente ritorna su *service1* dopo che il token SSO relativo è scaduto: la frequenza con cui viene chiesto all'utente di ri-autenticarsi è esclusivamente determinata dalla durata del cookie di sessione del servizio di login.

6.5.2 Servizi non-HTTP

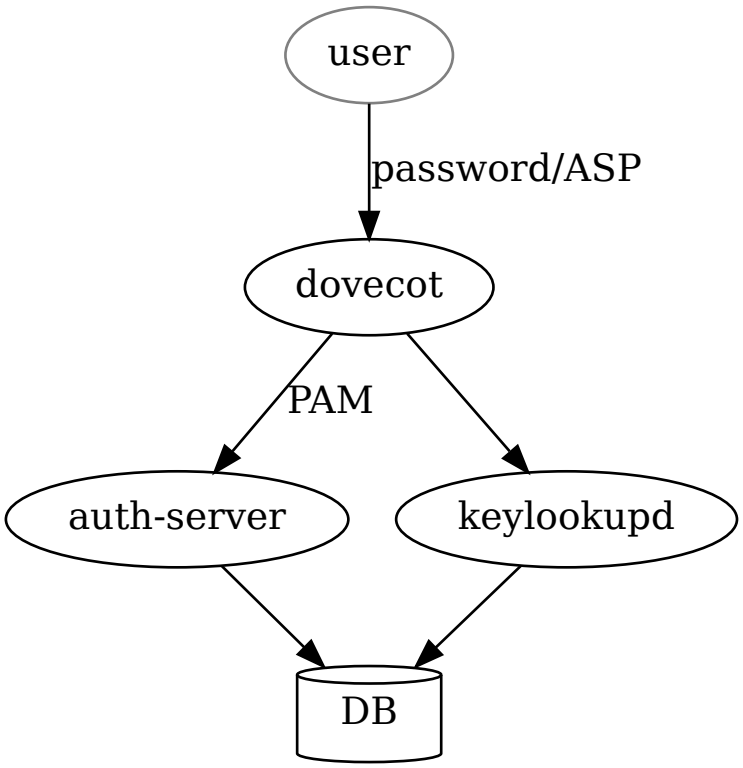

Figura 6.2: Flusso di autenticazione per Dovecot

Il caso di un servizio non web è molto più semplice perché in questo caso non c'è nessun secondo fattore di autenticazione, né un client in grado di mantenere uno stato come fa il browser con i cookies. Se l'utente ha 2FA abilitata, questo è il caso in cui si utilizza una application-specific password (ASP), che dal punto di vista del servizio è una semplice autenticazione a singolo fattore con username e password.

Vediamo il caso di *dovecot* (il servizio IMAP), che è interessante perché utilizza le credenziali crittografiche specifiche dell'utente:

1. il servizio riceve una richiesta di autenticazione con username e ASP
2. usando PAM, il meccanismo standard UNIX per l'autenticazione dei servizi (con il modulo *pam_auth_server*), Dovecot delega la verifica dell'autenticazione ad *auth-server*, che controlla le credenziali fornite con quelle presenti nel database
3. se l'autenticazione ha successo, Dovecot effettua una chiamata al servizio *dovecot-keylookupd*, che legge dal database la chiave crittografica dell'utente e la decifra con la password fornita.

6.5.3 Autenticazione per servizi terzi

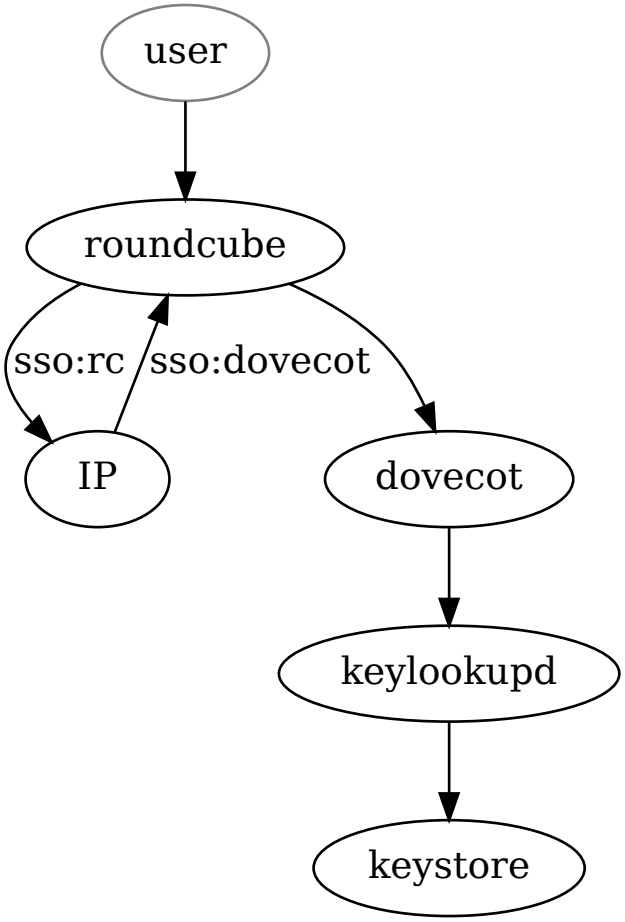

Figura 6.3: Esempio di trasferimento dell'autenticazione attraverso Roundcube e Dovecot

Un caso in cui i flussi di autenticazione descritti in precedenza si mischiano è quello in cui un servizio deve autenticarsi con un *altro* servizio interno, propagando però le credenziali dell'utente (dunque in aggiunta all'autenticazione *a livello di servizio* tra i due). Per esempio si può pensare alla *webmail*, dove l'applicazione web (Roundcube) deve autenticarsi con il servizio di posta impersonando l'utente.

1. l'utente visita la webmail
2. la webmail è un servizio HTTP protetto da Single Sign-On, quindi Roundcube possiede un token SSO valido per il servizio *webmail*
3. Roundcube fa una richiesta di *token exchange* al servizio di login, presentando il proprio token SSO valido e richiedendone uno valido per il servizio *dovecot*
4. il servizio di login verifica che la richiesta rientri nella lista ristretta di "scambi autorizzati", ovvero che sia possibile una transizione tra il servizio *webmail* e quello *dovecot*, e in caso positivo ritorna un nuovo

token SSO per l'utente originale, ma valido per il servizio *dovecot*
5. Roundcube fa una richiesta IMAP a dovecot, usando il token SSO come password
6. dovecot usa un altro modulo PAM (*pam_sso*) per verificare che il token SSO è valido
7. dovecot chiama il servizio *dovecot-keylookupd* passando la "password" che ha ricevuto, che però ricordiamo è un token SSO
8. un token SSO non è sufficiente per poter decifrare la chiave crittografica dell'utente contenuta nel database. Però nella descrizione del flusso di autenticazione "SSO" qui sopra abbiamo visto come in uno dei passaggi il servizio di login abbia salvato la chiave crittografica **decrittata** nel servizio *keystore*. Qui è dove viene utilizzata: dovecot-keylookupd invoca *keystore* per ottenere la chiave e ritornarla a dovecot.

6.6 Una API per il database delle utenze (*accountserver*)

Come spiegato in un capitolo precedente, il nostro database delle utenze ha una struttura complicata, ed in particolare necessita di una notevole quantità di logica di validazione esterna al database, sia per imporre criteri appartenenti alla "business logic" dei nostri servizi, che per mantenere l'ampio numero di dati denormalizzati.

L'*accountserver* è la nostra soluzione per fornire un'unica interfaccia centralizzata per la modifica dei dati sugli account contenuti nel database delle utenze, ed implementa tutta questa logica in un unico componente.

La motivazione per questa architettura deriva dalla nostra lunga (e un po' frustrante) esperienza con le strategie precedenti:

- la necessità di una API di alto livello, sopra al database stesso, per isolare le applicazioni dalla struttura del database, e fornirci astrazioni come *account* e *risorse* anziché soltanto oggetti annidati in LDAP o tabelle SQL;
- il desiderio di avere un'unica implementazione della *business logic*, e che ogni modifica al database passi attraverso di essa;
- l'esigenza di disaccoppiare logica e presentazione, per poter avere interfacce molteplici e separate (per esempio è utile, in termini di separazione di privilegi, che il pannello "utente" e quello di amministrazione degli account siano applicazioni separate).

Il servizio è implementato come un servizio RPC (*remote procedure call*). Riteniamo che questo approccio abbia diversi vantaggi:

- grazie alla centralizzazione, si possono più facilmente aggregare informazioni da backend multipli senza bisogno di duplicare logiche complesse lato client, e limitando la proliferazione di flussi RPC;
- separazione dei privilegi: accountserver è l'unico servizio autorizzato a scrivere sul database (controllato mediante ACL LDAP), tutti gli altri servizi hanno soltanto credenziali *read-only*;

- un servizio centralizzato offre un target più agevole per logging e monitoraggio.

L'interfaccia RPC è piuttosto estesa e comprende vari metodi specifici per la gestione di account (modifica di credenziali, recupero account tramite password secondaria, etc), ma in genere consente la manipolazione di due tipi di oggetti: *account* (*user*) e risorse (*resource*), contenute nei primi. All'account sono associati i parametri di autenticazione mentre le risorse rappresentano specifici account presso i vari servizi offerti, a seconda del loro tipo. Ciascun tipo di risorsa ha poi attributi differenti. Ogni account ha un identificativo primario che è un indirizzo email - conseguentemente per coerenza deve esistere, per l'account in questione, almeno una risorsa di tipo *email* con questo indirizzo. Altri tipi di risorse sono siti web, database, liste, etc.

All'atto pratico si è passati, dal punto di vista di un client, dal dover conoscere dettagli dello schema LDAP per poter creare una query opportuna (compresi tutti i filtri adatti a definire concetti complessi come "l'account è attivo"), all'utilizzo di una API RPC:

```
/api/user/get (username, sso_ticket)
```

che restituisce, presentando delle credenziali SSO valide per l'utente in questione, un oggetto JSON con informazioni sull'utente e su tutte le risorse ad esso associate, per esempio:


```
{
  "name": "user@example.com",
  "lang": "it",
  "uid": 19477,
  "status": "active",
  "shard": "3",
  "has_2fa": false,
  "has_otp": false,
  "has_encryption_keys": false,
  "resources": [
    {
      "id": "mail=user@example.com,uid=user@example.com,ou=People,dc=example,dc=com",
      "type": "email",
      "name": "user@example.com",
      "status": "active",
      "shard": "3",
      "original_shard": "3",
      "created_at": "2002-05-07",
      "usage_bytes": 412866884,
      "email": {
        "aliases": [
          "alias@example.com"
        ],
        "maildir": "example.com/user/",
        "quota_limit": 0
```

```
      }
    }
  ]
}
```

Le operazioni in scrittura sono nascoste dietro API esplicite, per esempio:

```
/api/user/change_password (username, sso_ticket, cur_password, new_password)
/api/resource/email/add_alias (resource_id, sso_ticket, alias)
...
```

È estremamente più semplice, per l'applicazione, operare su questi dati che sui risultati di query LDAP, e questo approccio consente di "nascondere" tutti gli orribili dettagli di implementazione del database dietro una API più elegante e mantenibile.

Dato che il carico in sola lettura su accountserver è relativamente elevato (ogni accesso al pannello utente implica una richiesta dei dati dell'account, per mostrare all'utente l'elenco delle sue risorse), e che comunque abbiamo un database LDAP locale ad ogni server, accountserver implementa un sistema *leader/follower* in cui eseguiamo istanze multiple di accountserver, ma una sola di esse è responsabile delle scritture al database: le altre istanze offrono un servizio in sola lettura, ed inoltrano eventuali richieste di scrittura all'istanza *leader*.

Esiste infine un'applicazione *accountadmin* per la gestione degli account, riservata alle admin, che consente di effettuare varie operazioni di alto livello (chiusura / apertura di un account, spostamento tra server, impostazione di varie opzioni, etc).

7 *Observability* del sistema

La possibilità di osservare, misurare ed analizzare in dettaglio il comportamento di un sistema è essenziale per ridurre il lavoro necessario a manutenere il sistema stesso: in particolare, la manutenzione richiede attenzione e tempo, le due risorse che più scarseggiano, e che dunque vanno spese con estrema cura. Abbiamo adottato una sequenza di approcci per affrontare questo problema, in ordine di priorità:

- Costruire sistemi *resilienti*, che non necessitino attenzione per eventi transitori e che ritornino autonomamente ad uno stato stabile. Per esempio avere molti *front-end* per ricevere il traffico significa che non ci dobbiamo preoccupare di problemi di rete presso i nostri provider.
- Avere sufficienti sistemi di controllo per essere avvisati quando c'è un problema reale, utili soprattutto a costruire la sicurezza che quando non ci sono allarmi tutto stia funzionando correttamente.
- Disporre di strumenti di analisi in grado di portarci rapidamente all'identificazione delle cause dei problemi ed alla loro risoluzione.

Questi ultimi due ruoli sono svolti, in cooperazione, dai servizi di *monitoraggio* e di *logging*.

7.1 Monitoraggio

Per monitoraggio (*monitoring*) s'intende lo studio delle caratteristiche di performance in tempo reale del sistema, derivate sia dall'osservazione di parametri interni al sistema (*whitebox* monitoring), che di comportamenti "esterni" (*blackbox* monitoring).

L'obiettivo del sistema di monitoraggio è raccogliere e conservare queste metriche, ed offrirci un linguaggio matematico per trattarle. Lo scopo è duplice: da una parte, ci deve permettere di valutare l'andamento delle metriche nel tempo, utilizzando strumenti analitici, usando i risultati come guida nella pianificazione e nella risoluzione dei problemi; dall'altra, ci consente di definire delle espressioni analitiche che esprimano la presenza di uno stato anomalo o di errore. Queste regole si chiamano *alert*: la loro definizione è una questione delicata, che merita un approfondimento.

7.1.1 Alerts

La nostra esperienza è che impostare un buon approccio all'*alerting*, ovvero decidere in quale modo portare all'attenzione delle componenti di A/I il fatto che qualcosa non funziona, è molto difficile ma anche

estremamente importante.

Per un progetto di volontariato, che non dispone dunque di una rotazione di reperibilità in grado di rispondere a problemi entro tempi definiti in qualunque ora del giorno e della notte, la prima cosa da fare è regolare correttamente le aspettative del pubblico: bisogna essere onesti e trasparenti e comunicare con attenzione il livello di prontezza nella risposta che le utenti possono ragionevolmente attendersi (che presumibilmente saranno lunghi).

Parte della soluzione è fare in modo che non sia generalmente necessario intervenire rapidamente in primo luogo, per esempio con la ridondanza dei servizi.

C'è un altro aspetto da considerare: in un contesto in cui nessuno viene pagata per investigare e risolvere problemi, è molto importante che il sistema di monitoraggio non sprechi il tempo delle persone. È dunque necessario che le alert siano:

- *Accurate*, ovvero che vengano generate solo quando c'è effettivamente un problema reale con implicazioni immediate per la qualità dei servizi offerti. In altre parole le alert devono avere un elevato rapporto segnale/rumore.
- *Informative* cioè devono puntare rapidamente l'operatrice nella direzione del problema.
- *Comprensive* nel senso che devono coprire nel modo più completo possibile i vari servizi offerti. Va da sé che è impossibile rilevare un problema là dove nessuno sta guardando.

Fallendo questi criteri ci si ritrova con un sistema di monitoraggio inutile e frustrante, che prima o poi verrà ignorato. Fortunatamente la tecnologia può aiutare a costruire invece un sistema che riesca a minimizzare sia il disservizio per le utenti che lo sforzo richiesto agli operatori.

Abbiamo basato il nostro monitoring sul principio dell'*alerting sintomatico*, ovvero sull'idea che le alert devono necessariamente corrispondere ad un evidente e dimostrato disservizio in atto per le utenti. Generalmente questo approccio si contrappone all'alerting "causale", in cui si controllano vari parametri interni presumendo che corrispondano a potenziali cause di problemi. L'approccio sintomatico riconosce che in un sistema distribuito complesso, questa inferenza è spesso impossibile e genera una notevole quantità di rumore.

Vediamo di fare un esempio per chiarire la differenza. Si consideri una qualsiasi applicazione web: possiamo avere alert per condizioni come "la CPU è usata al 100%" oppure "lo spazio disco è finito", e innumerevoli altre condizioni di questo genere, e implicare che se una di queste condizioni è vera, presumibilmente l'applicazione non sta funzionando a dovere. L'approccio sintomatico invece preferisce controllare condizioni come "il sito serve più dell'1% di errori HTTP" oppure "il 90mo percentile della latenza è maggiore di 100 millisecondi", che sono *direttamente* collegate all'esperienza dell'utente, ed assume che se queste condizioni sono false tutto il resto del sistema stia, per definizione, funzionando come deve. Il vantaggio del concentrarsi su ciò che è direttamente visibile dalle utenti è che non si rischia di "dimenticare" delle condizioni causali potenziali che potrebbero indurre problemi senza che lo sappiamo.

Definiamo quindi due tipologie di alert, distinte nello scopo:

1. Quelle che descrivono un problema sintomatico e richiedono l'immediata attenzione di un operatore. Le identifichiamo con il livello di severità *page*, ed attivano il processo di notifica che risulta nell'invio di email / messaggi / etc.
2. Le alert che riportano semplicemente uno stato anomalo, senza nessuna notifica associata. Queste coprono molto di ciò che normalmente si considera "alerting causale", e servono per la fase di debugging: avere una dashboard che riporta lo stato complessivo del sistema può aiutare a localizzare l'origine di un problema, una volta che si è ricevuta una notifica di una alert sintomatica.

Un altro obiettivo importante del sistema di alerting è quello di essere il meno verboso possibile pur mantenendo la propria efficacia: in teoria desidereremmo ricevere una singola alert per ogni "incidente". Non prestando attenzione a questo aspetto si corre il rischio di sovraccaricare chi opera con informazioni inutili e rumorose, anche in presenza di un effettivo problema, contribuendo allo stato di fatica cognitiva.

Il sistema di alerting che abbiamo adottato implementa una gerarchia standardizzata di alert che tiene conto della nostra conoscenza dei possibili *failure domains* (dominii di fallimento coordinato) del sistema: il risultato ci consente di ignorare, per esempio, alert "locali" in presenza di alert "globali" (es. non ci interessa sapere separatamente che ogni specifico server è rotto se lo sono tutti), oppure alert che si riferiscono ad un servizio su uno specifico server se il server in questione è irraggiungibile.

7.1.2 Blackbox monitoring

Uno degli investimenti che più hanno "pagato" in termini di utilità è stato la scrittura di una collezione di test programmabili per vari servizi, utilizzabili come *probe* in un contesto di blackbox monitoring. Questo software si chiama service-prober[1] ed è in grado di parlare diversi protocolli relativi a servizi che offriamo (HTTP, POP, IMAP, SMTP, etc.) ed eseguire test personalizzati.

Questo ci consente di mantenere sotto osservazione continua:

- la funzionalità del pannello utente e della webmail, compreso leggere / inviare messaggi di posta dalla webmail stessa;
- la capacità di ricevere ed inviare posta a server esterni;
- la funzionalità delle mailing list

ed altri comportamenti "complessi", usando utenti (sia interni ai nostri servizi, che esterni) dedicati esplicitamente allo scopo di test, ottenendo un'ampia copertura delle funzionalità dei nostri servizi. Avere la capacità di mantenere sotto osservazione questi parametri ci consente di verificare rapidamente eventuali segnalazioni di guasti, e capirne l'impatto, in termini che sono immediatamente rilevanti per gli utenti dei servizi stessi.

[1] https://git.autistici.org/ai3/tools/service-prober

Per di più, il software mantiene in memoria dei log completi di ogni transazione effettuata con i vari protocolli supportati, quindi quando si verifica un errore, è già disponibile un log dettagliato del problema.

7.2 Logging

Con *log* intendiamo l'insieme degli output diagnostici dei processi in esecuzione sulla nostra infrastruttura. Il termine comprende dunque una grande quantità di tipologie di dati differenti, dalla registrazione di eventi (come i log delle richieste HTTP) a messaggi diagnostici informativi. Alcuni di questi dati sono strutturati secondo schemi predefiniti, altri sono semplicemente testo libero. Se il monitoraggio presenta una visione aggregata della performance del sistema ("numero di richieste HTTP ricevute"), i log ne rappresentano la traccia dettagliata ("*queste* sono esattamente le richieste HTTP ricevute").

Proprio per via di questa informazione dettagliata che contengono, i log tendono ad includere anche vari tipi di dati potenzialmente delicati per la privacy dei nostri utenti, dunque vanno trattati con particolare cura e rispetto, come un materiale radioattivo: isolamento, smaltimento rapido, e soprattutto una "catena di custodia" esplicita e controllata.

D'altro canto queste informazioni diagnostiche dettagliate sono spesso essenziali per risolvere un problema, dunque c'è una tensione tra la necessità di mantenere log a scopo diagnostico, e l'eliminarli in quanto costituiscono un onere. Oltre a ciò, i log ed i dati in essi contenuti sono temi coperti da legislazione, e ci sono implicazioni legali da considerare.

Abbiamo deciso di risolvere questa tensione con l'*anonimizzazione*, ottenuta adottando accorgimenti a diversi livelli:

- Dove possibile, le informazioni problematiche vengono semplicemente *non incluse* nei log in primo luogo.
- I log vengono riscritti al volo, eliminando dati non voluti, *prima* di venire memorizzati su disco.
- Abbiamo stabilito un *orizzonte temporale*, eliminando i log più vecchi di un certo periodo di tempo (piuttosto breve).

Queste decisioni hanno ovviamente delle conseguenze, in primis l'impossibilità di debuggare problemi che siano avvenuti in passato oltre l'orizzonte temporale stabilito. All'atto pratico però questo risulta essere un impedimento minore di quel che si potrebbe pensare: in fin dei conti, è più importante essere in grado di risolvere un problema che si sta verificando *adesso* piuttosto che un problema che si è verificato in passato e che poi si è risolto da solo. Analogamente la mancanza per esempio di indirizzi IP nei nostri log non ha mai rappresentato davvero un problema, semplicemente perché in genere conoscere questi indirizzi non fornisce particolari informazioni diagnostiche aggiuntive: chiaramente ci sono meccanismi che hanno conoscenza di questi IP, per esempio i sistemi di rate-limiting delle richieste, ma li mantengono esclusivamente *in memoria*, e senza poter stabilire alcuna correlazione con le richieste effettuate dagli utenti.

Il sistema di logging da noi adottato ha dunque le seguenti caratteristiche:

- Offre una visione globale di tutti i log generati dalla nostra infrastruttura – otteniamo questo con un sistema centralizzato che raccoglie log dai vari processi sui vari server in un unico database (usiamo Elasticsearch per questo scopo).
- Ci offre un linguaggio per esprimere analisi e ricerche complesse, adatte alle molteplici tipologie di dati presenti nei log. Questo è un caso in cui utilizziamo volontariamente uno strumento *complicato* (Kibana): tool semplici come *grep* ed *awk* coprono senza dubbio il 90% dei casi analitici, ma per questioni di scala ci ritroviamo nella situazione di poter fare uso produttivo anche di funzionalità avanzate.

L'implementazione è piuttosto semplice, relativamente a queste cose, e si affida a *journald* per raccogliere i log da tutti i processi in esecuzione, il quale poi li inoltra ad un server *rsyslog* locale su ciascun host. I vari host inoltrano poi i loro log ad un server *rsyslog* di raccolta centrale, da dove vengono poi archiviati in Elasticsearch. L'unica caratteristica forse degna di nota è che in questo processo i log non toccano il disco fino all'ultimo passaggio, cosa importante visto che i nostri dischi sono lenti e molto utilizzati.

7.2.1 Estrazione di metriche dai log

A volte è utile poter estrarre metriche *real-time* direttamente dai log, per esempio quando si ha a che fare con software che non è in grado di esportare metriche direttamente. Due esempi comuni sono NGINX e Postfix, che non esportano metriche ma producono log con una traccia accurata di ogni transazione.

In questo caso utilizziamo mtail[2], uno di svariati programmi analoghi, che legge i log riga per riga, e utilizzando regular expression è in grado di derivarne delle metriche. Per esempio da una normale access log entry di NGINX può calcolare una metrica "numero di accessi HTTP" con tutti i campi di cui desideriamo tenere traccia come l'host, il metodo, lo status della richiesta, etc.

Quando si definiscono campi specifici per queste metriche, bisogna tenere conto della loro cardinalità (ossia il numero di valori distinti che possono assumere): i sistemi di monitoraggio, per loro natura, generalmente funzionano male con campi ad elevata cardinalità, per i quali è meglio affidarsi al database strutturato dei log, discusso nella sezione seguente.

7.2.2 Log come eventi strutturati

Tutti i log emessi dai nostri sistemi hanno una struttura, nel senso che seguono almeno lo schema minimale proprio di *syslog*, con attributi per host, *facility*, priorità, processo, etc. che ci consentono di filtrare accuratamente le ricerche.

[2] https://github.com/google/mtail

Altri log estendono questo schema: il nostro sistema di logging supporta la ricezione di eventi "strutturati" che altro non sono che oggetti JSON (usiamo uno standard un po' obsoleto ma estremamente semplice chiamato Lumberjack per trasportarli via syslog in modo trasparente).

Molti dei nostri log invece sono strutturati in un altro senso: semplicemente riportano informazioni in testo semplice ma con una struttura fissa (si pensi ai log HTTP). In questo caso ci conviene trasformare questa struttura fissa in uno schema esplicito, comprensibile al motore di ricerca dei log. Grazie ad una serie di regular expression, all'atto della raccolta finale dei log trasformiamo quindi cose come

```
noblogs.org - - [03/Nov/2022:10:59:18 +0000] "GET / HTTP/1.0" 200 ...
```

in (scritto in JSON per esemplificare):

```
{
  "vhost": "noblogs.org",
  "method": "GET",
  "uri": "/",
  "status": 200,
  ...
}
```

Lo scopo fondamentale di questi dati, al di là di facilitare le ricerche, è quello di poter generare report aggregati complessi utilizzando i metadati derivati (per esempio in delle *dashboard* web).

È bene sottolineare i diversi scopi del sistema di monitoraggio e del database dei log strutturati, perché specie nel caso di metriche derivate dai log, ci troviamo di fronte ad uno spettro di possibilità. Il sistema di monitoraggio fornisce dati in *tempo reale*, adatti ad informare un sistema di alerting con una visione globale della performance del sistema, economici da calcolare e memorizzare; il sistema di logging invece fornisce un database analitico adatto a rispondere a query complesse, costose da calcolare.

Prendiamo l'esempio dei log HTTP menzionati sopra, e consideriamo in particolare il fatto che il campo URL (ad alta cardinalità[3], controllato dall'utente) *non* è memorizzato nel monitoring: in questo caso, però il sistema di monitoraggio può rapidamente dirci, in tempo reale, quale frazione delle richieste in arrivo sta ritornando un errore 500, mentre il sistema di logging è più adatto a rispondere a query analitiche come "l'elenco delle 10 URL che hanno dato più errori in un dato intervallo di tempo".

[3] https://it.wikipedia.org/wiki/Cardinalit%C3%A0

8 Un esempio in dettaglio

8.1 Il sito web statico di A/I

Il sito web principale di A/I (https://www.autistici.org/) è un semplice sito dai contenuti statici che cambiano piuttosto di rado. Il codice sorgente è costituito da una serie di files Markdown, e da alcuni template HTML. La dimensione totale del contenuto è piuttosto limitata (< 1 GB). Associato al sito vi è anche un motore di ricerca, costituito da una piccola applicazione stand-alone scritta in Go.

Il processo di generazione delle pagine HTML però è alquanto elaborato, comprendendo l'applicazione dei template HTML alle pagine Markdown, la compilazione di tutto il codice Javascript, e la generazione dell'indice del motore di ricerca.

Per via di queste caratteristiche, abbiamo scelto di spostare tutta la complessità nel processo di *build* dell'immagine di container, che usa un processo multi-stage per ottenere alla fine un'immagine monolitica comprendente sia i contenuti che i servizi necessari a servirli. L'unica interfaccia pubblica del container è una porta HTTP (più una porta ausiliaria per il monitoraggio). Questo rende facile testare il container anche in locale senza particolari accorgimenti.

8.1.1 L'applicazione

L'applicazione web "sito di A/I"[1] è leggermente più complessa di un banale sito statico: intanto ci serve un web server avanzato (Apache o simili), al fine di usare *content negotiation* per redirigere le utenti al contenuto nella lingua corretta, e poi c'è una componente dinamica che deve servire i risultati della ricerca.

Utilizzando componenti che già abbiamo a disposizione, possiamo costruire un container con i seguenti processi:

- un server Apache che serve i contenuti statici via HTTP (ci serve in particolare Apache perché il sito utilizza pesantemente *content-negotiation* e *URL rewriting*), parte dell'immagine *apache2-base*[2];
- un exporter Prometheus per le metriche di Apache, parte anch'esso dell'immagine *apache2-base*;
- il server del motore di ricerca, che è in ascolto dentro il container su di una porta HTTP separata cui Apache inoltra le richieste per la URL `/search`.

[1] https://git.autistici.org/ai/website
[2] https://git.autistici.org/ai3/docker/apache2-base

Il container deve poi essere compatibile con il *runtime environment* di float, in particolare:

- deve tollerare l'esecuzione come qualsiasi utente non-root
- deve tollerare l'esecuzione con un'immagine base read-only

Fortunatamente per noi, l'immagine *apache2-base* già offre queste caratteristiche, quindi non dobbiamo fare niente di speciale. Oltre a ciò, questa immagine offre anche la possibilità di configurare alcuni parametri tramite variabili di environment, tra queste la porta su cui attivare il web server (`APACHE_PORT`). Questa possibilità è molto comoda in quanto ci consente di fare a meno di file di configurazione esterni al container.

Il `Dockerfile` per questo progetto assomiglierà dunque al seguente:

```
# Omettiamo i vari stage necessari a compilare gli asset statici,
# comprimerli, compilare il motore di ricerca e generare l'indice,
# cui fanno riferimento i "COPY --from" di seguito.

FROM registry.git.autistici.org/ai3/docker/apache2-base:master
COPY --from=gobuild /go/bin/sitesearch /usr/sbin/sitesearch
COPY --from=build /src/index /var/lib/sitesearch/index
COPY --from=assets /src/assets/templates/ /var/lib/sitesearch/templates/
COPY --from=precompress /var/www/autistici.org/ /var/www/autistici.org/
COPY docker/conf/ /etc/
RUN a2enmod headers rewrite negotiation proxy proxy_http \
    && a2dismod -f -q deflate \
    && chmod -R a+rX /var/lib/sitesearch /var/www/autistici.org
```

L'unico file nella directory *docker/conf* è la configurazione del virtualhost di default di Apache2.

8.1.2 Configurazione del servizio

Il container ha una *footprint* molto piccola, sia in termini di dimensioni del contenuto, che di risorse necessarie ad eseguirlo in produzione (trattandosi di un sito statico, Apache fa relativamente poco lavoro). Inoltre si tratta di un'applicazione completamente *stateless* quindi possiamo eseguirne molteplici istanze a scopo di *high availability* senza dover usare particolari accorgimenti, dato che l'infrastruttura che adottiamo è in grado di fare load balancing di traffico HTTP su istanze multiple.

Dal punto di vista dell'*orchestrator*, non abbiamo davvero bisogno di sapere dettagli su come il container funziona internamente. Tutto quel che c'è da sapere è:

- il container parla HTTP su una porta che possiamo controllare con la variabile d'ambiente `APACHE_PORT`;
- il container esporta delle metriche (via HTTP, sulla URL `/metrics`) su una porta differente, di default `APACHE_PORT` + 100.

La descrizione del servizio, nel formato usato da *float*, è dunque piuttosto semplice:

```
web-main:
  scheduling_group: frontend
  containers:
    - name: http
      image: registry.git.autistici.org/ai/website:master
      port: 8081
      env:
        APACHE_PORT: 8081
  monitoring_endpoints:
    - port: 8181
      scheme: http
  ports:
    - 8081
```

Abbiamo scelto la porta 8081 manualmente, sapendo che era disponibile (non utilizzata da altri servizi).

Manca una definizione di un *public_endpoint* che genererebbe automaticamente una configurazione per NGINX: per via del fatto che ospitiamo siti web come sottodirectory di autistici.org e inventati.org, questa configurazione è speciale e viene generata invece dai nostri script di automazione (vedi la sezione *Automazione*). Ma se non fosse stato così avremmo potuto aggiungere una sezione:

```
web-main:
  ...
  public_endpoints:
    - name: www
      port: 8081
      scheme: http
```

per avere automaticamente il sito raggiungibile come *www.autistici.org*, con i record DNS corretti e certificati SSL validi.

Dopo aver applicato la configurazione qui sopra in produzione, ci aspettiamo di trovare dei servizi systemd (sugli host appartenenti al gruppo *frontend*) chiamati *docker-web-main-http*, e infatti:

```
# systemctl status docker-web-main-http
* docker-web-main-http.service - web-main/http
     Loaded: loaded (/etc/systemd/system/docker-web-main-http.service; enabled;
     Active: active (running) since Sat 2022-09-03 15:51:25 UTC; 1 weeks 0 days
   Main PID: 2046356 (conmon)
      Tasks: 103 (limit: 19095)
     Memory: 109.8M
```

```
   CPU: 1h 7min 24.733s
CGroup: /system.slice/docker-web-main-http.service
        |-2046294 /usr/bin/podman run --cgroups=disabled --replace --sdnot
        |-2046356 /usr/bin/conmon --api-version 1 -c 6e4babea0b901a109c24b
        |-2046359 s6-svscan -t0 /var/run/s6/services
        |-2046385 s6-supervise s6-fdholderd
        |-2046464 s6-supervise sitesearch
        |-2046465 s6-supervise apache2
        |-2046466 s6-supervise apache-exporter
        |-2046469 /usr/sbin/sitesearch --index=/var/lib/sitesearch/index -
        |-2046470 /usr/sbin/apache2 -DFOREGROUND
        |-2046471 /usr/bin/apache_exporter -scrape_uri http://127.0.0.1:80
        |-2046545 /usr/bin/logger -p daemon error -t apache
        |-2046559 /usr/bin/logger -p local4 info -t apache
        |-2046560 /usr/sbin/apache2 -DFOREGROUND
        `-2046561 /usr/sbin/apache2 -DFOREGROUND
```

8.2 Il ciclo vitale di una modifica

Per capire come, all'atto pratico, si interagisca con un sistema di questo tipo possiamo esaminare il ciclo vitale di una ipotetica modifica al servizio appena esaminato (il sito web principale).

La prima fase consiste nella creazione della modifica in questione, e del suo upload sulla piattaforma di gestione del codice:

- Qualcuno fa una modifica al codice sorgente del sito: per esempio cambiando il logo del collettivo con una GIF animata di un gattino. Questa modifica viene committata in una nuova *branch*, chiamiamola *gatto*. Contestualmente alla branch, viene creata una *merge request* su git.autistici.org per consentire alle altre componenti del collettivo di commentare ed approvare la modifica.
- Il sistema di CI di git.autistici.org crea un'immagine Docker con il codice della nuova branch, che si chiama dunque `registry.git.autistici.org/ai/website:gatto`

8.2.1 Testing

La parte più importante della gestione di modifiche al sistema è la verifica della loro correttezza. La grande maggioranza del software che utilizziamo possiede una propria *test suite*, per verificarne a grandi linee il corretto funzionamento. Questo però non è lontanamente sufficiente a verificare che il risultato della combinazione

di tutto questo software, e delle nostre configurazioni, soddisfi le aspettative e produca dei servizi funzionanti. Per poter verificare ciò è necessario eseguire test di integrazione a livello di sistema.

Abbiamo dunque lavorato sulla capacità di creare ambienti di test a piacimento, utilizzando macchine virtuali (generalmente sul proprio PC), configurati con tutti o una parte dei nostri servizi in modo affine all'ambiente di produzione. La natura modulare della configurazione facilita questo setup: il sistema crea automaticamente tutte le credenziali necessarie, quindi è sufficiente un minimo di configurazione esemplificativa per avere una versione autonoma della nostra infrastruttura.

Oltre alla configurazione, è anche importante includere dei dati di test, per poter avere qualcosa (account utenti, siti web) da testare! Nel nostro caso abbiamo una serie di account utente di prova, con una password standard, più altri dati specifici dei vari servizi come siti web, blog di noblogs. Noblogs è forse il caso più rappresentativo dell'utilità di questo tipo di lavoro: anche se è stato necessario scrivere tool appositi per ottenere dump di specifici blog con tutti i dati anonimizzati, questo lavoro fa sì che possiamo testare manualmente le modifiche a Noblogs in modo particolarmente estensivo e completo, potendo facilmente confrontare il comportamento del nuovo software con quello effettivamente in produzione.

Venendo al caso in esame, adesso che esiste questa nuova immagine di container `:gatto`, è necessario verificare che sia corretta, non tanto in sé quanto nel contesto di tutta l'infrastruttura: la questione è semplice nel caso specifico di questo servizio che non ha dipendenze, ma si può facilmente immaginare l'importanza di questo passaggio in altri casi più complessi.

Per esprimere la volontà di utilizzare questa nuova immagine assieme a tutti gli altri servizi, bisogna modificare la configurazione dei servizi (*ai3/config*[3]) in modo da fare riferimento all'immagine:

```
web-main:
  ...
  containers:
    - name: http
      image: registry.git.autistici.org/ai/website:gatto
      ...
```

Con questa modifica, ci sono varie possibilità:

- Si può creare un ambiente di test a mano, in locale, con questa nuova configurazione, ed effettuare tutti i test manuali che si desiderano.
- Il repository ai3/config è configurato per eseguire una serie automatica di semplici test di funzionalità base dei servizi (posta, web) ad ogni commit. La CI tira su una serie di VM, installa un ambiente di test con le nuove modifiche, e lancia questa test suite integrata. I test non sono estremamente approfonditi, ma consentono di individuare rapidamente problemi grossolani, controllando cose come l'assenza di

[3] https://git.autistici.org/ai3/config

alert (quindi se tutti i servizi appaiono avviati correttamente), e la possibilità di autenticarsi ai vari servizi.

- Inoltre abbiamo aggiunto la possibilità di creare ambienti di test remoti, senza dover necessariamente lanciare il testbed sul proprio PC: utile per chi viaggia, o per chi ha computer poco potenti. È sufficiente creare una MR del repository ai3/config: in questo caso, Gitlab creerà automaticamente un ambiente di test su alcune VM temporanee (che si cancelleranno dopo un paio d'ore) per effettuare test manuali. Sarà possibile connettervisi tramite SSH o utilizzando un proxy SOCKS dedicato per testare il nuovo stato dei servizi.

8.2.2 Conclusione

Per affinare le modifiche fatte, con un processo iterativo, basta aggiornare la branch, e lanciare nuovamente l'automazione desiderata dell'ambiente di test. Questi passaggi tendono ad essere sufficientemente veloci anche se non sono completamente locali (ci si appoggia infatti in ogni caso a git.autistici.org per la distribuzione del container modificato).

Quando la modifica è approvata, la branch viene inclusa sulla branch principale (*master*), ed il prossimo *push* della configurazione aggiornerà il software in produzione.

Eventuali ambienti di test temporanei non sono più necessari e possono venire distrutti.

Si nota l'importanza sia di potersi appoggiare ad un sistema sufficientemente maturo di gestione del codice e di CI, per facilitare la collaborazione e come piattaforma di compilazione e di distribuzione dei container, sia il ruolo fondamentale degli ambienti di test per poter valutare gli effetti delle modifiche prima che raggiungano l'ambiente di produzione. Dato che l'ambiente di test è intrinsecamente affine a quello di produzione (con un po' di lavoro manuale occasionale necessario ad assicurarsi la presenza di *dati* di test sufficientemente rappresentativi), questo processo risparmia molta fatica e consente di agire con sicurezza.

9 Automazione

Questo capitolo tratta dell'automazione relativa agli account delle utenti, ovvero del seguente problema: una volta che abbiamo un'infrastruttura in grado di offrire servizi multi-utente, ed un database contenente l'elenco degli account su questi servizi, come colleghiamo le due cose? Per molti servizi la risposta è banale e comporta una connessione diretta: il servizio può semplicemente cercare di volta in volta i parametri di un account direttamente nel database. Non tutti i servizi però funzionano in questo modo, e in ogni caso la questione si complica se consideriamo anche i *dati* associati agli account.

In realtà quello che più spesso ci ritroviamo a dover fare è una *sincronizzazione* tra sistemi differenti: da una parte il database delle utenze, dall'altra un qualche database interno specifico del servizio, oppure (molto spesso) il filesystem.

Ci sono molti approcci possibili al problema della sincronizzazione, e quello che abbiamo scelto, in utilizzo dal 2005, riflette in parte le nostre capacità tecniche dell'epoca, in parte le tecnologie che erano allora comuni e disponibili: avevamo un database veloce e "facile" da replicare, mentre non erano ancora diffuse tecnologie *log-based*. In ogni caso crediamo che sia valido ancora oggi, per via delle sue caratteristiche di robustezza.

9.1 Macchine a stati

Modelliamo i compiti di automazione degli account come una serie di *macchine a stati finiti*[1], dove gli stati sono memorizzati nel database LDAP, e le transizioni sono attuate da una serie di *script* eseguiti con cadenza periodica. Di solito uno script si occupa di una particolare transizione, eventualmente ristretta ad un particolare tipo di account (posta, web, etc). Gli stati degli account sono indipendenti l'uno dall'altro dunque l'automazione può fare progressi per ciascun account indipendentemente, e il meccanismo si presta naturalmente alla parallelizzazione.

Il periodo di esecuzione di questi script determina il "battito" di tutta l'automazione, determinando la velocità del processo di sincronizzazione, con cui i cambiamenti si propagano dal database ai vari servizi. Per il modo in cui è strutturata la nostra automazione, questo non rappresenta un limite una volta chiaro a noi stesse ed alle utenti che nessuna modifica è istantanea e che c'è da aspettarsi un tempo di propagazione non nullo.

Consideriamo in dettaglio, per esempio, il processo di migrazione dei dati di un account (per un servizio partizionato, ovvero dove i dati di un account sono associati ad un particolare server) dal server A al server

[1] https://it.wikipedia.org/wiki/Automa_a_stati_finiti

B. Questo workflow utilizza due attributi dell'oggetto associato all'account nel database LDAP, *host* e *originalHost*:

- All'inizio, l'account ha sia *host* che *originalHost* impostati ad A
- Il workflow viene avviato (manualmente, o da altri processi automatici) cambiando il valore dell'attributo *host* a B
- Uno script periodico sul server B nota un account con *host* uguale a B ma *originalHost* diverso da B, e comincia a copiare i dati dell'account dal server indicato da *originalHost*, ovvero A
- Quando questo script termina con successo, imposta il valore dell'attributo *originalHost* a B
- Uno script periodico sul server A nota che l'account ha sia *host* che *originalHost* diversi da A, dunque cancella i dati associati all'account.

L'account passa dunque attraverso questi stati:

- *host=A*, *originalHost=A*
- *host=B*, *originalHost=A*
- *host=B*, *originalHost=B*

L'account è in uno stato ben definito a ciascun passaggio: a seconda dello specifico servizio si potrà poi scegliere se servire l'account seguendo l'attributo *host* (l'account apparirà temporaneamente "vuoto" e verrà riempito con il backfill dei dati dal server A), l'attributo *originalHost* (i dati rimarranno accessibili ma si perderanno i dati modificati durante la migrazione), oppure semplicemente sospendere il servizio per l'account finché *host* è diverso da *originalHost* (approccio utile a mantenere l'integrità dei dati rispetto alle modifiche, al prezzo di una indisponibilità temporanea).

Va notato che l'implementazione di macchine a stati descritta è adatta ai compiti semplici che ci servono, ma diventa rapidamente scomoda con il moltiplicarsi degli stati.

9.2 Riconciliazione

Con le semplici macchine a stati descritte sopra implementiamo un approccio alla sincronizzazione detto di *riconciliazione*: assumendo che il database rappresenti la copia primaria ed autorevole dei dati, ovvero esprima *l'intenzione* di configurazione dei servizi, noi confrontiamo il database con lo stato effettivo del sistema, e compiamo operazioni per eliminare le differenze riscontrate.

Questo approccio è spesso contrapposto alla cosiddetta *task queue*, ovvero la soluzione per cui una operazione da effettuarsi su un account è rappresentata come un *task* che verrà eseguito nella sua interezza non appena possibile.

L'approccio riconciliatorio ha dei vantaggi che lo rendono appetibile nel nostro caso:

- È un'architettura intrinsecamente robusta e *fault-tolerant*: le macchine possono essere irraggiungibili, i processi possono fallire, e comunque eventualmente il sistema convergerà verso lo stato desiderato, facendo progressi ogni volta che ne sarà in condizione.
- L'attività degli script è completamente locale ed isolata, dunque questi possono essere semplici e comprensibili.

Ci sono ovviamente degli svantaggi, a cominciare dal fatto che un approccio *eventualmente consistente* implica l'esistenza di stati intermedi potenzialmente *inconsistenti*: dato che manipoliamo stati molto semplici (una mailbox o riceve posta o non la riceve) questo si manifesta principalmente come il ritardo di propagazione tra l'espressione dell'intenzione (modifica al database) e la sua effettiva applicazione, e non introduce problemi nel servizio. Altri stati inconsistenti si possono verificare laddove sia necessario un coordinamento: per esempio, tra la configurazione del reverse proxy HTTP e quella del servizio associato, che sono configurati indipendentemente. Anche questo risulta non essere un grandissimo problema: dato che transizioni di questo tipo occorrono o all'atto della creazione di una nuova risorsa, o in occasione di problemi, una indisponibilità temporanea della risorsa associata è difficilmente notabile.

Infine c'è il fatto che ogni iterazione dell'automazione richiede una scansione completa sia del database che dello stato attuale del servizio, operazione che può essere costosa su grandi volumi (ma non ancora ai nostri livelli, c'è qualche ordine di grandezza di margine). Uno dei modi per rimediare a questo problema, dovesse diventare importante, è ridurre la frequenza di esecuzione, che però implicherebbe un corrispondente rallentamento della velocità di propagazione.

Il principale svantaggio di questo approccio è la confusione creata dalla lunghezza del ciclo di *feedback* tra l'azione ed il suo effetto: il fatto che dopo aver per esempio modificato il database delle utenze poi non succeda assolutamente niente per decine di minuti suggerisce immediatamente un problema in ciò che si è fatto. È necessaria della formazione per introiettare questo meccanismo, il che rappresenta un grosso ostacolo, compensato soltanto dalla notevole robustezza e stabilità offerta dall'approccio riconciliativo.

9.3 Generatori di configurazioni

Si è menzionato come alcuni servizi richiedano una configurazione esplicita per ciascun account ed altri no. Consideriamo due esempi specifici per chiarire di cosa si parla:

- Ad un estremo dello spettro di configurabilità stanno servizi come il *web hosting*, dove dobbiamo creare una configurazione separata per ciascun sito presente nel nostro database. Apache è un software molto flessibile e potente, con una superficie di configurabilità estremamente ampia, che richiede dunque una descrizione molto verbosa di quello che deve fare.
- All'altro estremo ci sono servizi come IMAP e XMPP, dove invece l'unica specificità dell'utente è la sua identità: qui possiamo cavarcela con una configurazione globale del servizio, e un collegamento diretto al database delle utenti.

Un certo numero dei nostri script di automazione è dunque dedicato alla pura generazione di configurazioni per servizi complessi, generalmente utilizzando *template* specifici. Quando questi script rilevano un cambiamento nelle configurazioni generate, riavviano automaticamente il servizio associato.

Il trade-off tra una configurazione statica ed una dinamica è anche a volte una questione di complessità e performance: per fare un esempio, è effettivamente possibile, con l'ausilio di moduli particolari, configurare Apache in modo "dinamico", prendendo l'elenco dei siti ospitati direttamente da un database. Questo però introduce una dipendenza diretta dal database, senza il quale il servizio web non può più funzionare, mentre nel caso statico un fallimento del database impedisce soltanto la propagazione di cambiamenti nella configurazione. C'è anche da considerare che una configurazione dinamica di Apache richiede un accesso al database per ogni richiesta HTTP, che può diventare un problema con un alto volume di traffico. Dove siano possibili entrambe le tipologie di configurazione, di volta in volta è dunque necessario considerare anche:

- la frequenza con cui cambiano le configurazioni, ed il costo del *reload* di un servizio - se la frequenza è molto elevata è preferibile una configurazione dinamica;
- l'adattabilità dei modelli di dati: se è necessaria della logica applicativa per adattare i dati del database a quelli desiderati dall'applicazione, può essere più semplice esprimerla in un *template*.

9.4 Manipolatori di account e dati

Dato che nella maggior parte dei casi alle risorse del nostro database sono associati dei *dati*, una buona parte della nostra automazione è dedicata alla loro gestione, sincronizzando lo stato delle risorse nel database con il filesystem (o altri database specifici dei vari servizi).

In particolare questi compiti di gestione si raggruppano in tre categorie:

- *Creazione* di risorse: alcuni servizi (es. web hosting) hanno bisogno che le relative directory sul filesystem esistano, con permessi specifici, prima di funzionare correttamente. Per altri servizi (es. mailing list) è necessario creare entità nei database specifici del servizio.
- *Cancellazione* delle risorse non più desiderate: quando una risorsa viene cancellata, eventualmente vogliamo che anche i files associati vengano rimossi dal filesystem.
- *Spostamento* dei dati da una partizione ad un'altra. Questa categoria di script implementa la macchina a stati per lo spostamento delle risorse tra server descritta in precedenza, copiando dati dal server "vecchio" e modificando il database. All'atto pratico, dove non sia disponibile un protocollo specifico per il servizio (es. MySQL per copiare i database) la copia viene effettuata usando un servizio *rsync* interno dedicato a questo scopo.

I primi due compiti sono in genere accorpati assieme in script che utilizzano un banale algoritmo di *differenza di insiemi* per trovare gli account da creare e quelli da eliminare in un unico passaggio:

- elenca gli account presenti effettivamente sul sistema

- elenca gli account nel database
- crea gli account che sono nel database ma non sul sistema
- cancella gli account che sono sul sistema ma non nel database

La rimozione dei dati è un processo rischioso in quanto irreversibile, dunque per le risorse pubbliche come i siti web adottiamo una cautela ulteriore: prima di cancellare i dati dal filesystem, se si tratta di dati pubblici, li archiviamo su un servizio di backup dedicato (cosiddetto *cold storage*) che ne conserva una copia a lungo termine.

9.5 Riconciliazione inversa

Ci sono alcuni casi in cui il database delle utenze *non* è autoritativo per tutti i parametri di una risorsa. Questo generalmente avviene quando un servizio ha una sua interfaccia per modificare questi parametri che ci interessano, e non abbiamo potuto o voluto modificare il software per parlare direttamente con il nostro database. Ci sono un paio di questi casi:

- L'elenco degli amministratori di una mailing list può essere modificato dagli stessi attraverso l'interfaccia web di Mailman. Questo è un workflow nativo in Mailman e vorremmo continuare a supportarlo, per non confondere troppo le utenti (o, peggio, dover mantenere delle *patch* al codice di Mailman).
- Le password degli utenti MySQL possono essere cambiate da dentro MySQL medesimo. Caso meno comune, ma che comunque sarebbe bene supportare.

Per gestire questa situazione abbiamo dell'automazione che effettua una "riconciliazione inversa": questi dati vengono letti dal database solo all'atto della *creazione* della nuova risorsa, mentre periodicamente vengono sincronizzati nella direzione opposta, dal servizio al database.

L'importante, in questi casi, è riconoscere che questi dati sono nel database delle utenze solo a titolo informativo, ed in "sola lettura": per riprendere l'esempio delle mailing list (Mailman), avere gli amministratori della lista nel database non significa che possiamo cambiarli lì, ma ci consente invece di mostrare le liste nel pannello utente, fornendo l'illusione di un servizio *trasparente* all'utente finale.

10 Alla prova del tempo

Sono già passati un paio d'anni da quando il collettivo A/I ha adottato l'infrastruttura descritta in questo documento, quindi un minimo di analisi in retrospettiva è già possibile. Nel complesso diremmo che ha ottenuto i risultati desiderati: il livello di manutenzione si è dimostrato basso come previsto, le operazioni "forzate" (ad esempio il passaggio ad una nuova versione stabile di Debian) sono state effettuate con scadenze indipendenti e senza fretta, etc.

La separazione dei segreti dal codice ci ha anche permesso finalmente, dopo tanti anni che lo desideravamo, di pubblicare *tutto* il codice e la configurazione dei nostri servizi, cosa che speriamo sia stata utile ad altri gruppi che fanno cose simili, e che ricordiamo di nuovo, si trova qui: https://git.autistici.org/explore/

Abbiamo tratto grande beneficio dalla possibilità di testare modifiche arbitrarie all'infrastruttura, migliorando Noblogs in modo significativo, e consentendo ad alcuni di noi di sviluppare competenze specifiche al riguardo.

Lentamente vediamo anche come le tecnologie "professionali" si stiano allineando nelle direzioni che ci servono, tanto da poter abbandonare vari strumenti e soluzioni *custom* in favore di equivalenti free ed open-source: siamo probabilmente vicine al momento in cui potremo anche sostituire *float* stesso.

11 Outro


```
"Socializzare saperi, senza fondare poteri."
-- Primo Moroni
```

[1] https://creativecommons.org/licenses/by-nc-sa/4.0