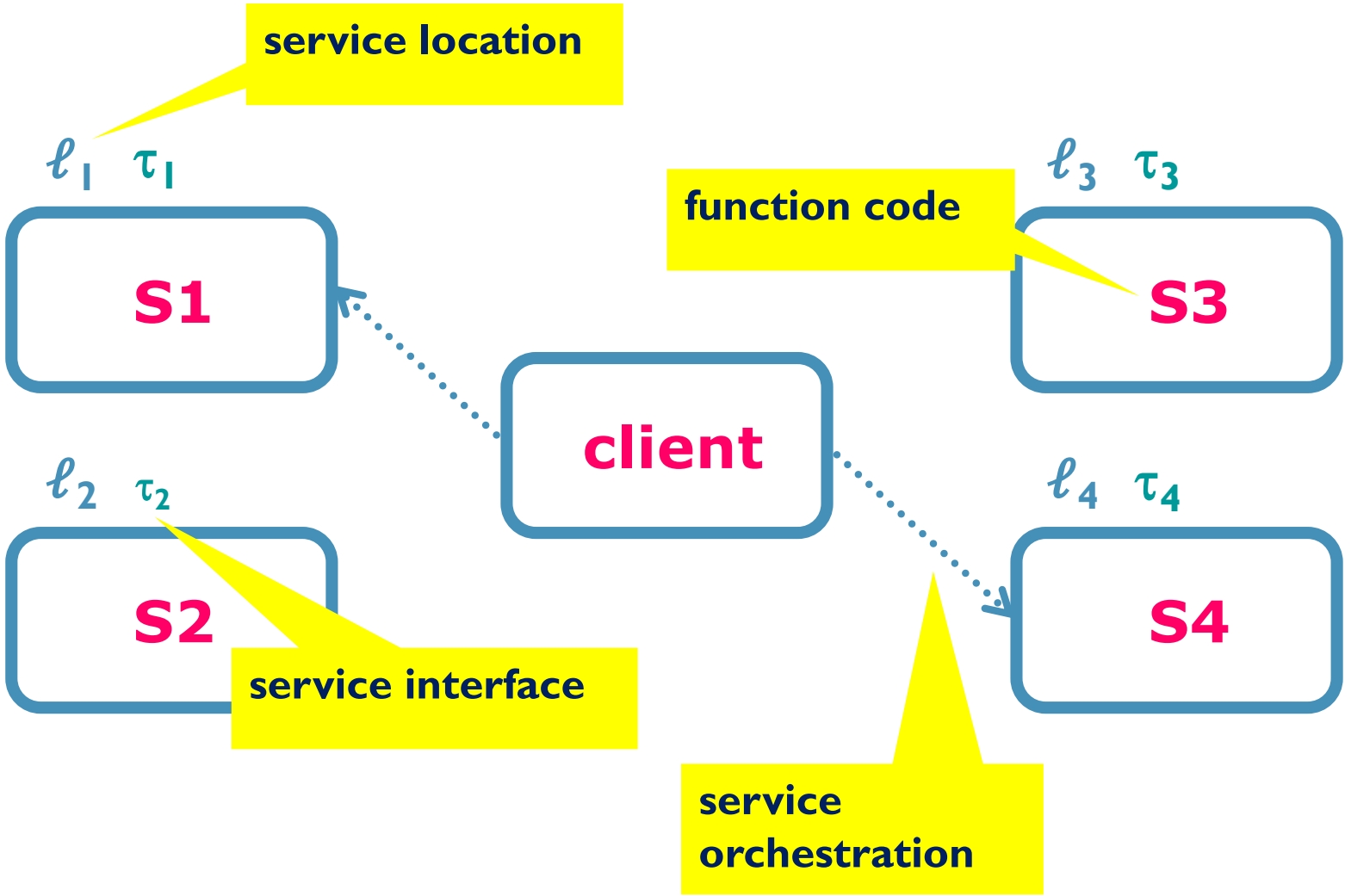
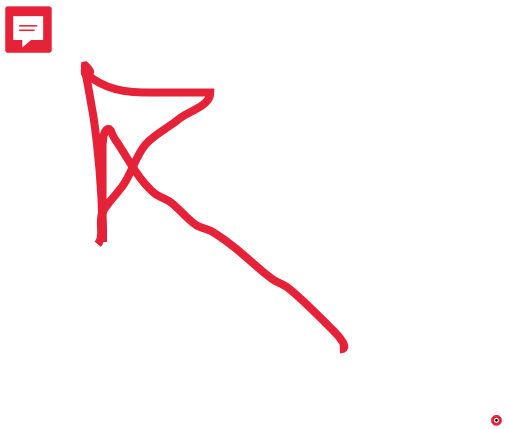
AMALGATING STATIC AND DYNAMIC ENFORCEMENT OF SECURITY PROPERTIES

PROGRAMMING IN A WORLD OF STATELESS SERVICES

Ci addentriamo in un contesto in cui vogliamo affrontare alcuni aspetti della sicurezza cercando di riprendere e di usare delle tecniche che abbiamo visto separatamente. Il caso di studio ha a che vedere sia con linguaggio di programmazione e sia con le applicazioni distribuite in modo particolare supponiamo di essere in un mondo di servizi stateless come voi, forse avete visto in altri corsi o in generale avete letto, ora uno dei trend che hanno i fornitori di servizi sull'IoT cloud continuum è quello di avere dei servizi che sono sostanzialmente delle funzioni senza stato, si chiama usando una buzzword è la FaaS, ed è il primo fornitore di servizi che ha portato questa tecnologia nell uso nella pratica comune è amazon con il sistema di AWS della piattaforma LAMBDA. L'idea è che un cliente vuole utilizzare dei servizi che sono come delle funzioni, vuol dire che io chiamo la funziona, mi restituisce un risultato, quindi sono dei servizi stateless che non mantengono lo stato, cioè tra due invocazioni successive della stessa funzione, non viene mantenuto lo stato dell'invocazione precedente, quindi si può rappresentare con questo disegnino: ho un cliente, questo cliente sa che ad un certo punto nel mondo c'è qualcuno che ha messo a disposizione dei servizi e l'idea è che di ogni servizio uno sa la locazione o in altri termini, chi è il fornitore del servizio e dove questo fornitore di servizio ha fatto il deployment del proprio servizio, quindi quello che nel disegno è chiamato l1, poi per ogni servizio metto a disposizione un'interfaccia del servizio che dice quali sono le caratteristiche del servizio e in modo particolare nel nostro esempio vedremo soltanto gli aspetti di natura funzionale nel caso poi di AWS lambda e dei fornitori servizio, c'è tutto un aspetto anche che è presente nell interfaccia che ha vedere col meccanismo di billing, ovvero quel meccanismo di pagamento dei servizi offerti e poi esiste l'altra cosa che chiaramente il cliente non vede, il codice, S3 cioè il codice servizio che viene programmato e poi esiste questo meccanismo, si chiama orchestrazione. L'orchestrazione è che il cliente definisce un passo di esecuzione normalmente sono dei linguaggi di programmazione molto semplici in cui fa vedere come vengono chiamati i servizi e l' ordine di sequenza dei servizi, sostanzialmente l'orchestratore è la la visione del cliente che dice prima chiamo il servizio l1, poi una volta che ho chiamato il servizio l1 chiamo il servizio l2 e l3 oppure chiamo in parallelo il servizio l2 al servizio l1 poi faccio un operazione che collega i risultati di l1. Se si vedono i linguaggi che i vari fornitori di servizio forniscono sono sostanzialmente dei linguaggi tipo while con quindi dei linguaggi con la possibilità di fare dei cicli e poi di avere del parallelismo senza sincronizzazione iniziale, ma con una sorta di join, quindi una specie di parallelismo fork & join anche se sintatticamente non esiste la nozione di join. Ci mettiamo in una situazione dove vogliamo definire l' orchestratore prendendo la visione del cliente, vogliamo definire un meccanismo che ci permette di invocare i servizi che vengono offerti e di garantire al massimo la sicurezza.



- Two kinds of security concerns:
  - secrecy of transmitted data, authentication, etc (**network security**)
  - control on computational resources (**access control, resource usage analysis, information flow control, etc**)
- need for linguistic mechanisms that:
  - **work in a distributed setting**
  - **assume no trust relations among services**
  - **can also cope with *mobile code***



SECURITY AND SERVICE COMPOSITION

IN-LINE MONITOR: SAFETY FRAMINGS

allora abbiamo detto che cerchiamo di mettere assieme alcuni aspetti che abbiamo visto in isolato allora la prima cosa che noi andiamo a vedere è come definiamo all'interno del linguaggio dei meccanismi che hanno esattamente la stessa strategia dell'inline reference monitor cioè che va a controllare dei pezzi di codice, solo che lo vediamo esattamente come primitiva linguistica, cioè quando noi abbiamo visto l'inline reference monitor lo abbiamo visto con un meccanismo di descrizione e poi con un meccanismo che ci permetteva di fare un'operazione di instrumentazione del codice a partire dalla proprietà che volevamo. Quello che noi invece ora vediamo è proprio un meccanismo linguistico definito all'interno del linguaggio di programmazione che concettualmente stiamo sviluppando, che ci permette di andare a controllare delle porzioni di codice

IN-LINE MONITOR: SAFETY FRAMINGS

Client: protect from untrusted code

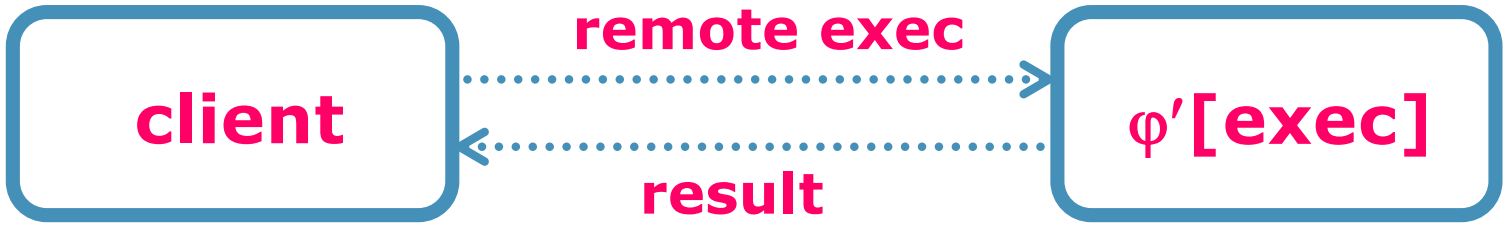
In-line monitor: safety framing

The **policy φ** is enforced stepwise within its **scope**

SECURITY AND SERVICE COMPOSITION: SAFETY FRAMINGS

Services: protect from clients

(now the **safety framing** belongs to the service)

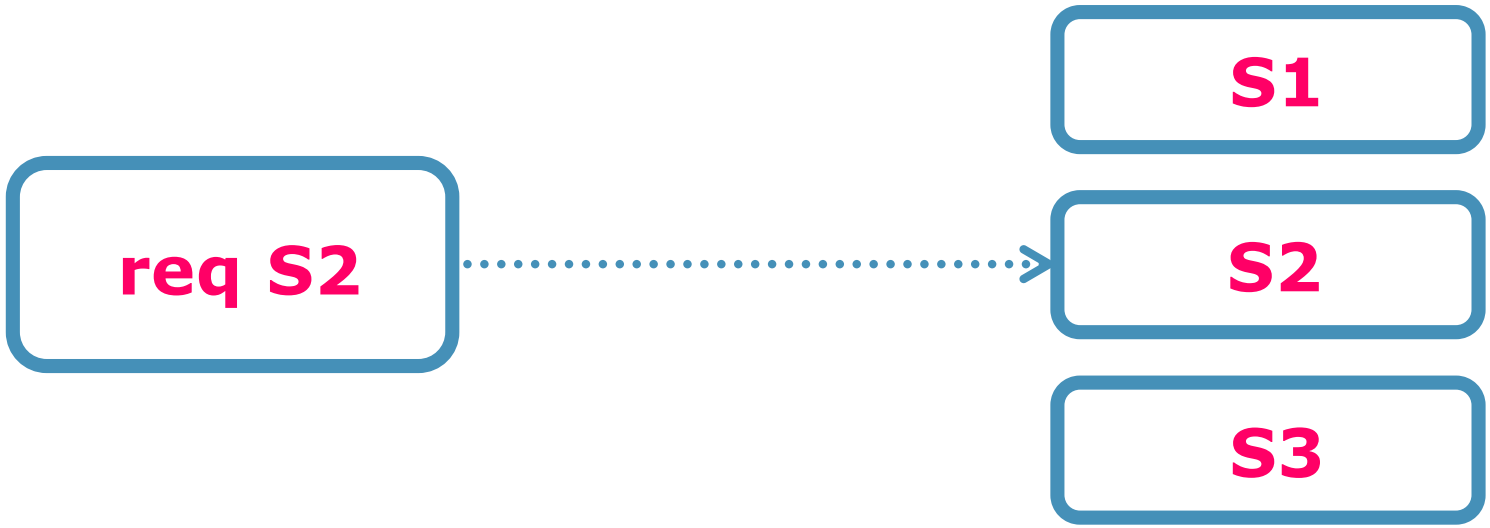


Scoped policies check the **local execution histories**

SERVICE SELECTION

a questo punto, una volta detto che è come affrontiamo il problema del controllo, andiamo a esaminare un altro aspetto che ha a che vedere su come viene individuato un servizio. Abbiamo tre servizi s1 s2 e s3, supponiamo che questi tre servizi siano sostanzialmente equivalenti, vuol dire abbiamo tre fornitori dei servizi oppure lo stesso fornitore di servizio che ci offre tre versioni differenti dello stesso servizio e il cliente fa una richiesta che in questo momento scriviamo in questo modo e poi andremo a vedere meglio nel dettaglio come lo scriveremo fa la richiesta del servizio s2 quello che succede è che noi dobbiamo fare un binding a tempo di esecuzione nell infrastruttura tra il punto in cui avviene la richiesta dell' invocazione del servizio ed effettivamente il codice servizio. Quello che succede è che abbiamo sostanzialmente un meccanismo, che ci permette di invocare il servizio remoto, passargli il parametro e prendere i risultati questo è quello che succede nel retrobottega dell' implementazione. Vorremmo essere un po' più tolleranti rispetto al servizio, ad esempio supponendo proprio esattamente che s1 s2 e s3 sono servizi equivalenti che cosa succede ad esempio, se il fornitore del servizio s2 è troppo carico. Se noi abbiamo fatto il binding una volta per tutte tra la nostra richiesta ed s2 se il fornitore del servizio è troppo carico, noi siamo in attesa, quindi abbiamo un meccanismo di downgrading delle prestazioni dovuto al troppo overhead a tempo di esecuzione. Dato che operiamo in un ambiente altamente dinamico dove è difficile prevedere quello che succederà a runtime, vogliamo avere dei meccanismi che ci garantiscono un maggiore adattabilità alle caratteristiche del contesto di esecuzione quindi vogliamo che un binding di una richiesta di servizio al servizio si possa adattare il più possibile alle caratteristiche attuali di quando questo servizio verrà invocato.

Req-by-name: request a *given* (function as a) service among many



Why **S2** and not **S1** or **S3**, if all functionally equivalent ?

Abbiamo individuato il servizio, gli diamo il nome del servizio dove col nome del servizio si tende molto di più del nome si intende il numero e locazione, meccanismo di invocazione e il pattern di invocazione. Il punto di questa osservazione è che noi vorremmo passare da un meccanismo basato sul nome a un meccanismo contestuale, cioè un meccanismo che tiene conto anche di tutte informazioni di natura non funzionale che hanno a che vedere con l' invocazione del servizio. Un esempio che abbiamo fatto prima è il carico a runtime in quel momento del sistema, altre sono il livello di protezione. Per esempio se s1 s2 s3 sono tre versioni differenti dello stesso fornitore di servizio una gold un platinum e una bronze, dove questo medaglione vogliono definire il livello di sicurezza dei vari servizi, allora ci potrebbero essere delle situazioni in cui uno vuole un livello di sicurezza alto, quindi platino o situazioni in cui vuole un livello di sicurezza di bronzo. Uno vorrebbe avere un meccanismo dove l' invocazione del servizio non dipende dal nome, ma dipende dalle proprietà che si aspetta dal fornitore del servizio, quindi vuole avere un passaggio da un meccanismo dove io invoco un servizio e determino la discovery del servizio in base al nome del servizio, a un meccanismo che invece faccio la discovery e il bind del servizio in termini della proprietà che mi aspetto e notate che questa cosa qui è quello che sintatticamente e con un certo livello di astrazione succede anche quando io ho i puntatori alle funzioni virtuali. Io posso mettere un puntatore, un codice di una funzione e quindi questo puntatore può essere associato a diversi target ovvero diversi codice, proprio perché il puntatore a un codice è un puntatore che poi può essere associato a diverse attuazioni della funzione. Allora noi vogliamo fare questa stessa operazione, non dandogli semplicemente uno smart pointer che poi prenderà valori diversi e quindi associata a insieme di target intesi come funzioni ma vogliamo fare questa operazione dove invece di avere la nozione di pointer abbiamo la nozione di proprietà, quindi facciamo un invocazione di una funzione in base alla proprietà che il fornitore del servizio deve rispettare. Quindi quello che qui è lo slogan, vogliamo passare da un'informazione di natura sintattica a una informazione di natura semantica che tiene conto di tante informazioni di contesto che quella sintattica non può offrire.

Problems with "request by name":

- what if named service **S2** becomes unavailable ?
- …and if **S2** is outperformed by **S1** or **S3** ?
- hard reasoning about non-functional properties of services (e.g. security)
- security level independent of the execution context (unless hard-wired in the service code)

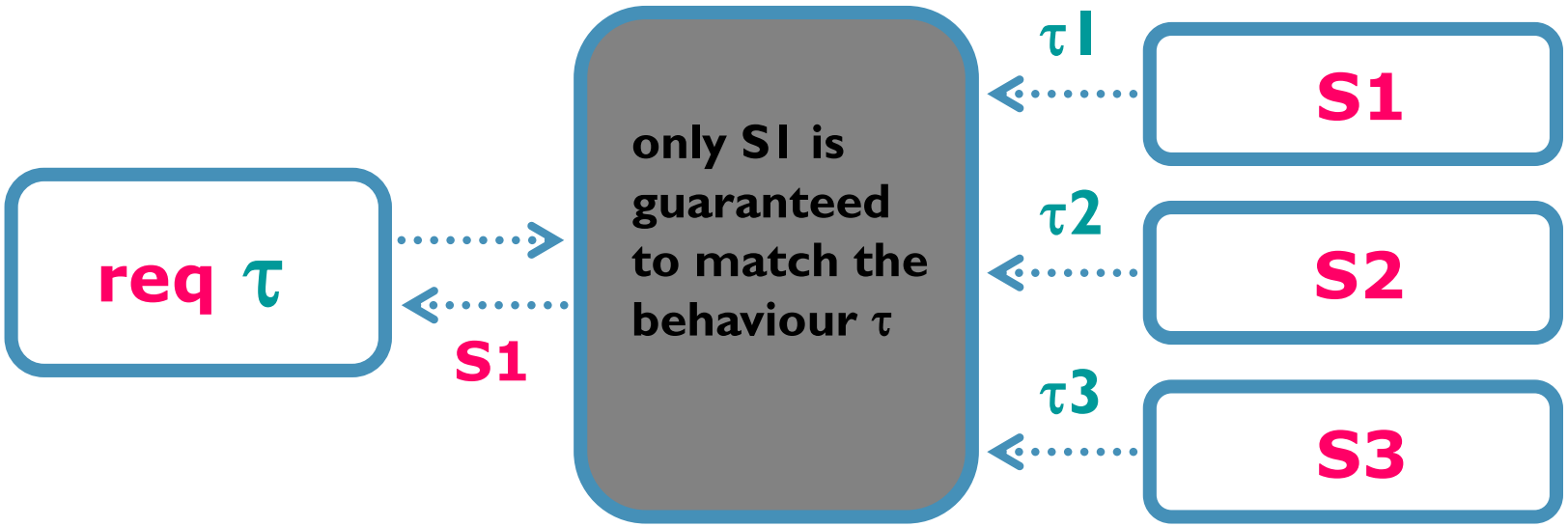
From syntax-based to semantics-based invocation

Service names $\ell, \ell', ..$ tell me nothing about the behaviour!

Vorremmo realizzare una invocazione per contratto cioè abbiamo un contratto d'uso, io cliente ho diversi fornitori di servizio con i fornitori di servizio ho definito un pattern per scrivere dei contratti, allora a questo punto il contratto è quello che mi guida nell'invocazione quindi dal lato del cliente a livello di programma non scrivo il nome della funzione ma dico il contratto, ovviamente nel linguaggio dei contratti che ho stabilito con i miei fornitori di servizio, il linguaggio di contratti che devono essere rispettati, quindi in modo particolare con TAU indicherò il contratto che deve essere rispettato tipicamente il contratto sarà un qualcosa della funzione che mi aspetto, cioè gli argomenti in ingresso, gli argomenti in uscita e l'eventuale proprietà di sicurezza che mi aspetto dalla funzione stessa. Il contratto mi dirà anche se voglio utilizzare delle azioni sulle risorse, quali azioni sulle risorse voglio utilizzare in modo tale anche da vedere quali sono i vincoli di sicurezza delle azioni che voglio fare sulle risorse. Ricordate che sin dall'inizio ho detto che uno degli aspetti che vogliamo considerare è l'aspetto che ha a che vedere col controllo dell'uso delle risorse. A questo punto il cliente vede solo il fornitore del servizio, quindi c'è una barriera di astrazione che nasconde al cliente il fornitore del servizio, quindi il fornitore del servizio non è noto, effettivamente quello che vede è soltanto il meccanismo di selezione dei servizi e vede che i vari servizi s1 s2 e s3 si sono messi d'accordo di fornire il servizio tau2 tau3. A questo punto, il selezionatore selezionerà il servizio "ottimo" tra tau1-2-3 rispetto alle richieste che fa tau vuol dire che il contratto di servizio metterà non solo delle informazioni di natura funzionale I/O ma tutte le informazioni di contesto che sono requisiti non funzionali. Noi in questo contesto andremo a vedere soltanto i requisiti non funzionali soltanto la sicurezza in un altro esempio andremo a vedere aspetti di trusted e in un terzo esempio ancora andremo a vedere aspetti che hanno a che fare con l' overhead a tempo di esecuzione, quindi è un meccanismo che si presta a diverse interpretazioni

Req-by-contract: request a service respecting

the desired behaviour



τ imposes both functional and non-functional constraints

USE CASES FOR REQ-BY-CONTRACT

Example: download code that obeys the policy φ

Example: a remote executer that obeys the policy φ'

OBSERVABLE BEHAVIOUR

Dal punto di vista che ora prendiamo il comportamento osservabile sono le azioni che sta facendo sulle risorse, quindi tutte le operazioni che sto facendo sulle risorse che sono significative per il livello della sicurezza, ad esempio leggere scrivere dei file, invocare o no una certo operazione su un database o su altri particolari servizi e così via, quindi sono quelli che abbiamo qui chiamato gli eventi di accesso, cioè sono delle informazioni che ci dicono che stiamo accedendo una certa risorsa con una qualche operazione e sono programmabili, quindi supponiamo che a livello del linguaggio di programmazione io ho un meccanismo di programmare questi accessi, vuol dire che deve avere un interfaccia globale tra i clienti e i fornitori di servizio dove si dichiarano quali sono le azioni disponibili e in che modo queste azioni sono disponibili. Il significato è definito globalmente all'applicazione che stiamo considerando, ma c'è un aspetto, quindi queste sono esattamente i primi due punti quindi essendo programmabili, possono essere inserite nel programma e quindi derivate da un analizzatore statico del programma e ha un significato globale all'interno del sistema che stiamo considerando, ma un punto importante che ha a che vedere con la sicurezza è che non possono essere nascoste vuol dire che una qualunque entità che si occupa di descrivere e di vedere il comportamento allo stato attuale dell'esecuzione non può nasconderli nel retrobottega quindi vuol dire che sono degli eventi che sono globalmente accessibili quando uno vuole osservare il programma, allora a questo punto qual è il comportamento astratto? E' dato da quelle che noi abbiamo chiamato execution history che sono quindi sequenze di azioni che hanno a che vedere con le risorse di sicurezza che sono accessibili a run time, quindi dal punto di vista dell'osservazione della stabilità dei programmi, quello che noi andiamo osservare dei programmi sono le azioni che fanno su le risorse di sicurezza, le azioni di sicurezza che avevano fatte sulle risorse non vediamo altro dell'esecuzione del programma.

- **access events** are the actions relevant for security (e.g. read/write local files, invoke/be invoked by a given service, etc)
  - mechanically inferred, or inserted by programmer.
  - their meaning is fixed globally.
  - access events cannot be hidden.
- the (abstract) behaviour observable by the orchestrator over-approximates the **histories**, i.e. sequences of access events, obtainable at run-time.

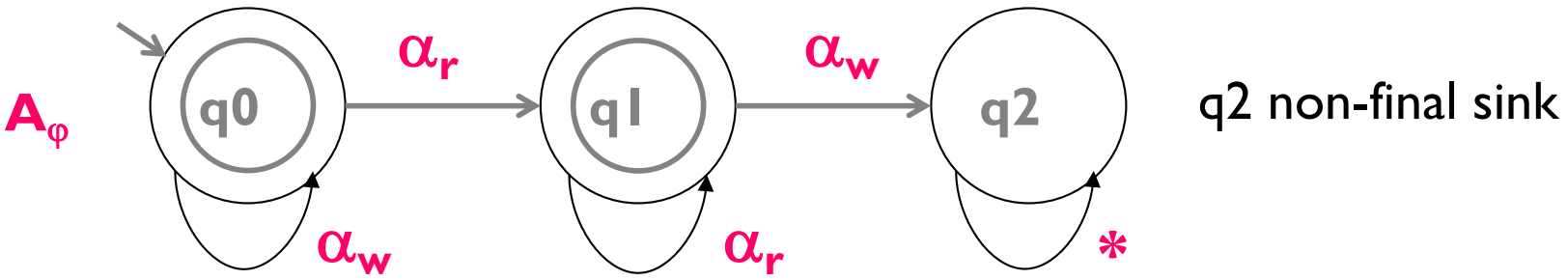
WHAT KIND OF POLICIES ?

Andiamo a vedere quali sono le politiche che noi vediamo, allora la nostra idea è che le politiche sono delle proprietà regolari sulle storie di esecuzione. Sostanzialmente vuol dire che noi abbiamo un espressione regolare che ci permette di scriverla oppure un semplice automa, quindi gli automi e le espressioni regolari voi sapete che sono un modo di descrivere delle proprietà di safety, cioè ad ogni passo dell esecuzione nel programma noi vogliamo garantire un certo grado di sicurezza, l'altra cosa che noi vogliamo fare è definire delle politiche che possono essere innestate uno dentro l'altro perché abbiamo detto che le politiche noi le vediamo associate a i blocchi che possono essere innestati uno dentro l'altro, le politiche possono essere ovviamente innestate una dentro l'altra. Abbiamo parlato di esecuzione e l'esecuzione sono tutte esecuzione locali non è che abbiamo una nozione di esecuzione in un contesto distribuito, quello che noi vediamo localmente ad ogni sito, ad ogni cliente, ad ogni orchestrazione, vediamo il suo flusso di esecuzione.

- History-based security
- Policies φ are **regular** properties event histories
- Policies φ, φ' have a **local scope**, possibily **nested** $\varphi[\cdot\cdot\varphi'[\cdot\cdot]\cdot\cdot]$
- A policy can only control histories of a **single** site (no trust among services)
- Histories are **local** to **stateless** service sites (stateful easy)

EXAMPLE: CHINESE WALL POLICY

φ Chinese Wall: cannot write (α_w) after read (α_r)



$\alpha_w \alpha_r \alpha_w \not\models \varphi$

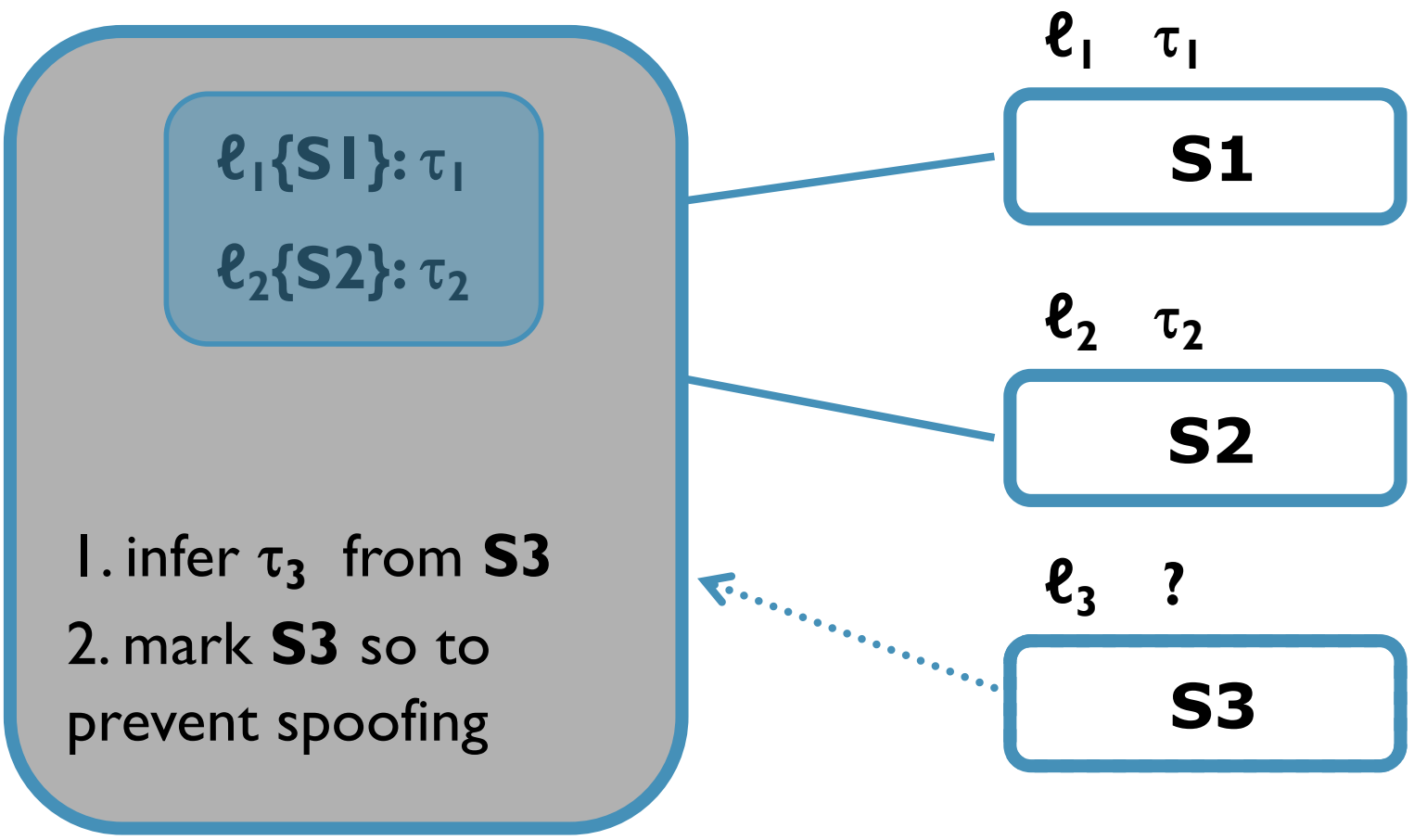
PRINCIPLE OF LEAST PRIVILEGE

"Programs should be granted the minimum set of rights needed to accomplish their task"

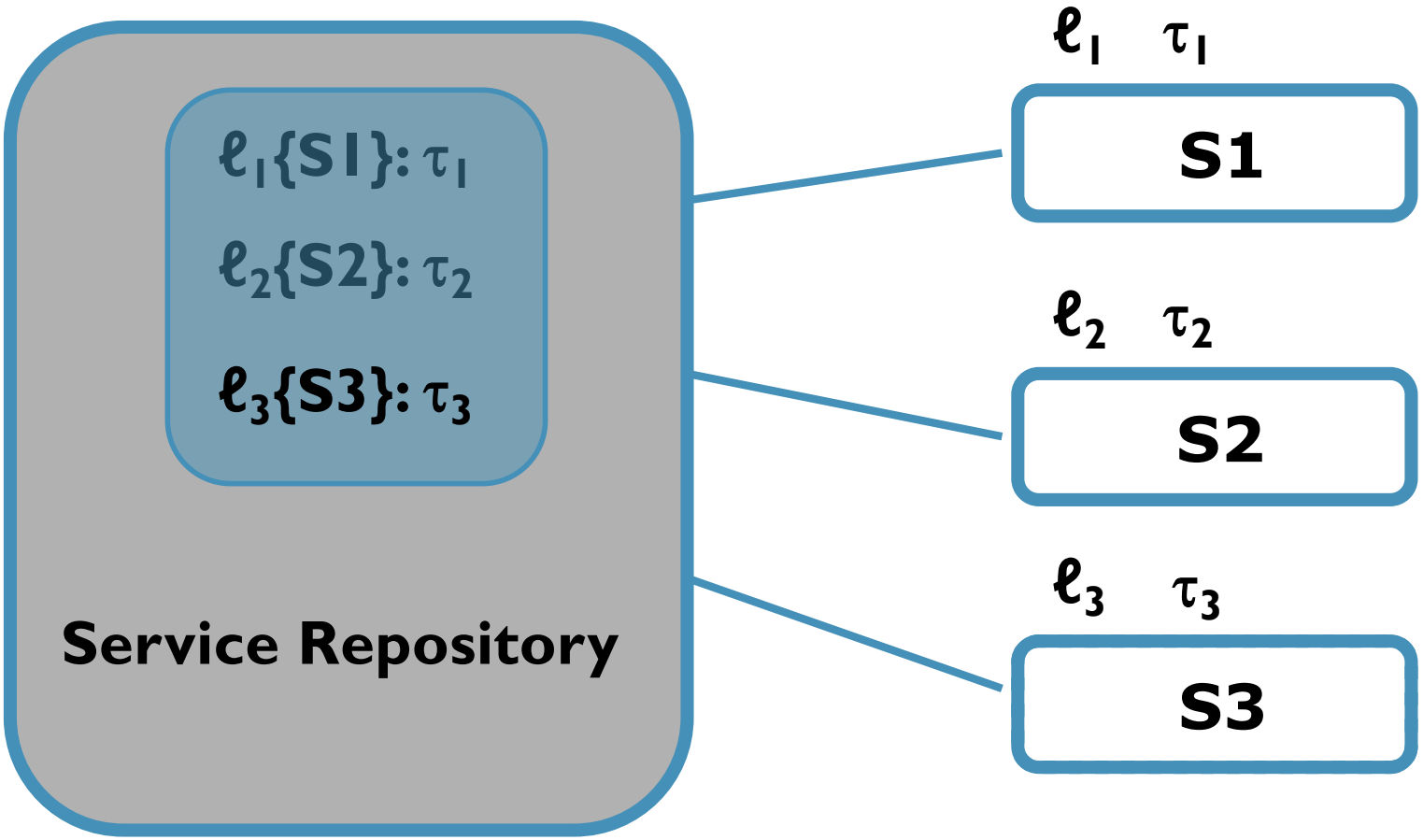
- A service must always obey all the active policies (no policy override)
- Policies can always inspect the whole past history (no event can be discarded)
- "Privileged calls" implemented by policies that explicitly discard the past

andiamo a vedere come viene fatta la pubblicazione del servizio. Supponiamo che io sono un fornitore del servizio, sono alla locazione l3 e voglio pubblicare un servizio con un codice s3. Supponiamo che altri hanno già pubblicato il servizio s1 e s2 con le loro informazioni di contratto. Il meccanismo che pubblica i servizi assume che il fornitore sta alla locazione l3 ma non solo prende il codice, va anche a inferire, quindi vuol dire che facendo un'analisi del codice va a inferire il contratto di comportamento, quindi il comportamento del servizio s3 e quindi è il tipo tau3 che viene inferito vuol dire che bisogna mandare il codice, e uno lo può fare anche mandando il codice e il tipo, ma poi sicuramente il fornitore del servizio va a controllare che il tipo dichiarato è effettivamente quello inferito, andando a fare un'analisi dei binari e queste sono dei meccanismi che succedono in tanti compilatori si chiama PROOF CARING CODE. La pubblicazione del servizio si inferisce il codice ma si fa anche una marcatura crittografica del codice s3 in modo da garantire che non ci sia dello spoofing, cioè dici che uso quello, ma in realtà quello è un fake poi in realtà viene utilizzato un altro codice, quindi questo serve per avere un minimo livello di affidabilità della applicazione. Adesso nella directory dove sono pubblicati i servizi disponibili ci sarà che l3 è come codice S3 marcato con il contratto tau3 qualunque questo sia.

SERVICE PUBLICATION (1)

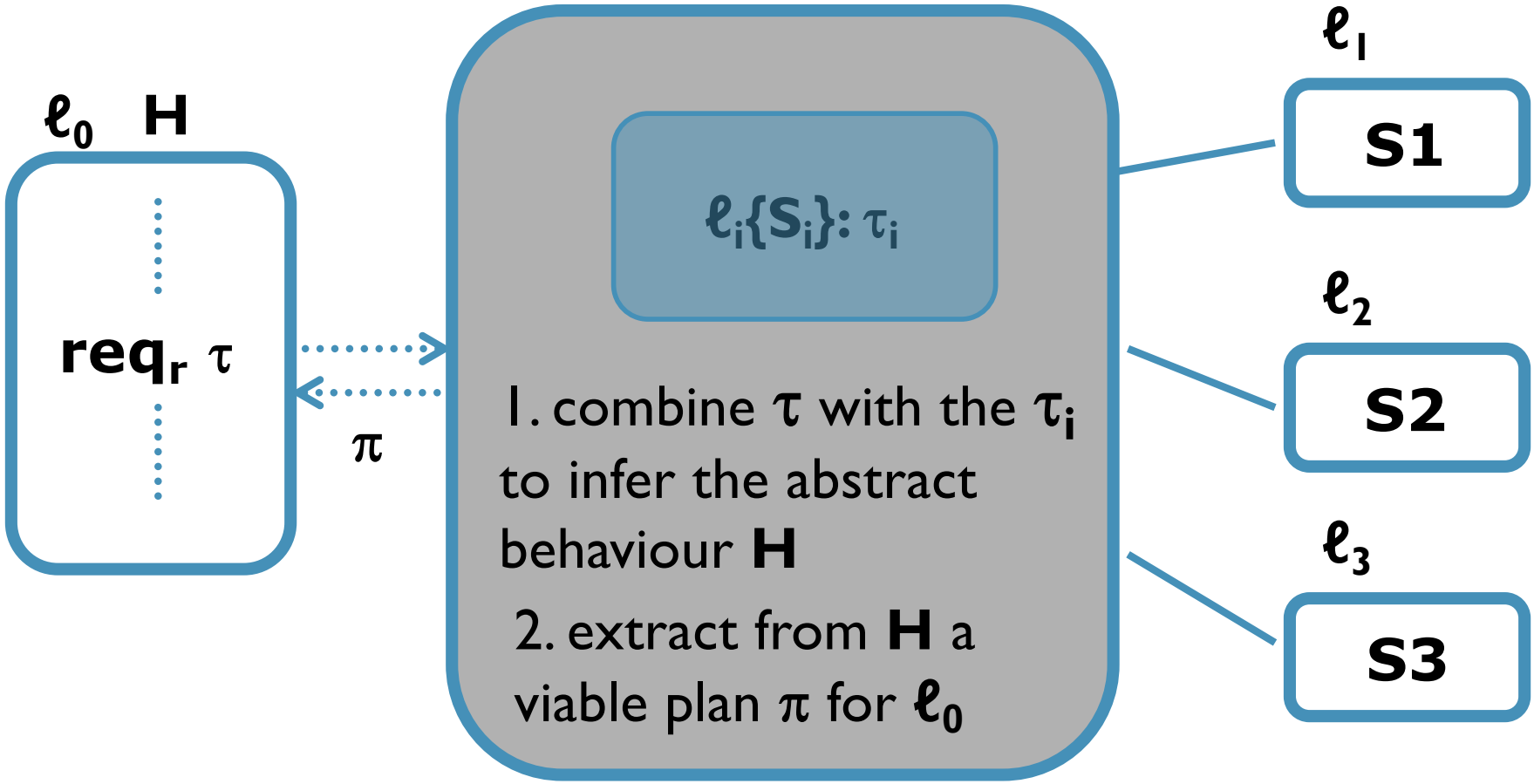


SERVICE PUBLICATION (2)

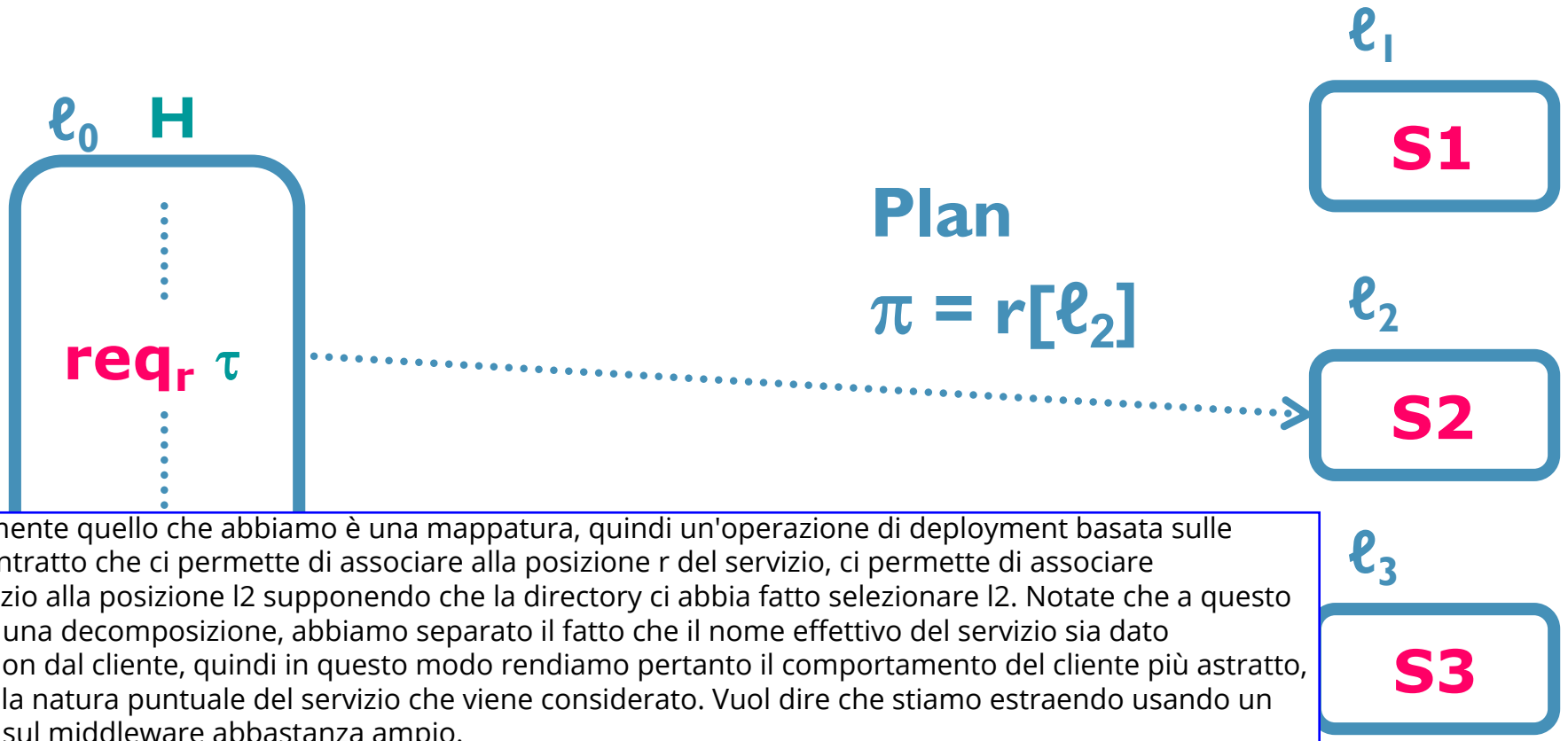


A questo punto l' orchestratore del servizio fa una richiesta del contratto, quindi fa una richiesta di un contratto tau, la cosa che deve fare il fornitore dei servizi, quindi la directory del servizio è andare a vedere quale tra i tre contratti è compatibile con la richiesta del servizio e quindi deve fare un'operazione che ha a che vedere esattamente con la struttura dei codici per inferire, usando dei sistemi di tipo, il codice astratto e poi a questo punto, da questo codice astratto dell' invocazione riesce a inferire quello che noi abbiamo chiamato il piano, cioè il meccanismo di play sync alla mappatura che associa alla richiesta di invocazione di un servizio tau visto come contratto l' effettivo codice a quella specifica locazione che verrà mandata in esecuzione.

SERVICE ORCHESTRATION



SERVICE ORCHESTRATION



Quindi concettualmente quello che abbiamo è una mappatura, quindi un'operazione di deployment basata sulle informazioni del contratto che ci permette di associare alla posizione r del servizio, ci permette di associare esattamente il servizio alla posizione l2 supponendo che la directory ci abbia fatto selezionare l2. Notate che a questo punto noi abbiamo una decomposizione, abbiamo separato il fatto che il nome effettivo del servizio sia dato dall'orchestratore non dal cliente, quindi in questo modo rendiamo pertanto il comportamento del cliente più astratto, non dipendente dalla natura puntuale del servizio che viene considerato. Vuol dire che stiamo estraendo usando un livello di astrazione sul middleware abbastanza ampio.

Names are only known by the orchestrator!

la mappatura può considerare diversi aspetti, ad esempio, che non si devono mai violare le politiche che non ci devono essere delle richieste del servizio che non sono risolte, che uno può in un certo punto dire che non vuole più il monitor in esecuzione. Uno può avere delle mappature che tengono conto delle alternative quello che abbiamo chiamato i multi choices, cioè che ad una certa richiesta posso avere più scelte, poi è il cliente a fare la scelta, oppure scelte dipendenti, vuol dire se prima ho fatto una certa richiesta dove ho preso un servizio basato su una particolare architettura, poi questo mi vincolerà successivamente nelle richieste di altri servizi, cioè il meccanismo di mapping dalla visione astratta alla visione concreta sul middleware di esecuzione ha diversi stadi di riferimento noi faremo vedere in questo momento quello più semplice, ma ci sono diversi modi di affrontarlo.

WHAT IS A PLAN ?

- A plan drives the execution of an application, by associating each service request with one (or more) appropriate services
- With a **viable plan**:
  - executions never violate policies
  - there are no unresolved requests
  - you can then dispose from any execution monitoring!
- Many kinds of plans:
  - **Simple:** one service for each request
  - **Multi-choice:** more services for each request
  - **Dependent:** one service, and a continuation plan
  - ...

WHO DO WE TRUST ?

The orchestrator, that:

- certifies the behavioural descriptions of services **(types annotated with effects H)**
- composes the descriptions, and ensures that selected services match the requested types
- extracts the **viable plans** (through model-checking)

Also, someone must ensure that services do not change their code on-the-fly

Un minimo livello di trust di affidabilità lo assumiamo in modo particolare noi assumiamo che i tipi che ci danno il comportamento con gli effetti di esecuzione sono certificati, cioè se un fornitore del servizio e la directory del servizio mi dice che il fornitore del servizio ha messo a disposizione un qualcosa che ha un codice con un certo tipo, noi ci fidiamo del tipo in modo particolare perché i contratti sono quelle cose che poi noi utilizzeremo per modellare il mapping e per modellare il meccanismo di astrazione del mapping sulla particolare architettura. ovviamente c'è un terzo meccanismo che noi per il momento abbiamo detto mettiamo una marcatura tipo spoofing in quello che noi assumiamo e che qualcun altro, cioè l'architettura, deve in un qualche modo assumere con meccanismi del livello delle architetture che uno non può cambiare nel retrobottega al codice, una volta che l'ha pubblicato.

SUMMING UP…

a programming model for secure service composition:

- **distributed** stateless services
- **safety framings** scoped policies on localized execution histories
- **req-by-contract** service invocation

static orchestrator:

- certifies the **behavioural interfaces** of services
- provides a client with the **viable plans** driving secure executions

WHAT'S NEXT

- programming: syntax and **operational semantics**
- static semantics: **type & effect system**
  - **types** carry annotations H about service behaviour
  - **effects** H are history expressions, which over-approximate the actual execution histories
  - **Combination of EM and Static Analysis**
- extracting viable plans:
  - **linearization:** unscrambling the structure of H
  - **model checking:** valid plans are viable