POLITECNICO DI MILANO

FACOLTÀ DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea in Ingegneria Informatica

LA VERSION CONSISTENCY NELLA RICONFIGURAZIONE DINAMICA DEI PROCESSI DI BUSINESS

Relatore: Prof. SAM JESUS GUINEA MONTALVO
Correlatore: Ing. VALERIO PANZICA LA MANNA

Tesi di Laurea di:
DAMIANO BONETTI, matr. 750199
STEFANO CICERI, matr. 751630

Anno Accademico 2011 - 2012

Indice

Capitolo 1

Introduzione

I sistemi distribuiti basati su componenti (CBDS, Component-based distributed systems) costituiscono un modello architetturale che sta avendo una rapida diffusione nel mondo IT: l'utilizzo dei grandi sistemi centralizzati sta calando rispetto alla crescente necessità di architetture composte da componenti diversificati e separati tra loro.
Non esiste un modello unico di CBDS, proprio perchè ogni sistema distribuito ha caratteristiche hardware (macchine e strutture di interconnessione) e software (linguaggi, strutture dati) specifiche per il contesto in cui è calato; l'eterogeneità è quindi una caratteristica ben evidente di questa tipologia di sistemi. Tuttavia è possibile delineare alcune definizioni comuni: l'architettura di un CBDS può essere definita dalla *configurazione statica*, un grafo orientato in cui i nodi costituiscono i componenti del sistema e gli archi definiscono le possibili interazioni tra di essi; i componenti possono eseguire *transazioni*, che rappresentano una sequenza di azioni eseguite e completate in un certo lasso di tempo, e che includono sia computazioni locali che scambi di messaggi.

I CBDS spesso devono reagire a continui cambiamenti del mondo in cui sono calati e nei requisiti che devono soddisfare. Questi cambiamenti sono spesso difficili da predirre a design time o troppo costosi da identificare e da gestire dal software inizialmente progettato. Per adattarsi ai cambiamenti dei requisiti e dell'ambiente circostante, il software deve essere in grado di evolvere. L'evoluzione del software viene comunemente supportata attraverso il rilascio di patches o upgrades. L'installazione di aggiornamenti software è tipicamente effettuata offline e richiede l'arresto ed il riavvio del sistema. Tuttavia, diverse categorie di sistemi non possono interrompere le transazioni in esecuzione per effettuare l'aggiornamento: l'interruzione di sistemi bancari o finanziari provocherebbe ad esempio una perdita economica significativa, mentre l'arresto di sistemi critici potrebbe essere un rischio per i propri utenti. Per queste tipologie di sistemi il software deve essere in grado di evolvere dinamicamente modificando le parti esistenti dell'implementazione ed aggiungendo nuove funzionalità a runtime, senza

fermare l'intero sistema.
Rispetto agli aggiornamenti offline, le modifiche dinamiche non sono banali poichè oltre alla correttezza della nuova versione, devono garantire il corretto completamento delle transazioni che il sistema sta eseguendo. Per *correttezza dell'aggiornamento* di un componente intendiamo che le specifiche dell'intero sistema devono essere sempre soddisfatte dalle transazioni in corso prima, durante e dopo l'update.
Gli studi sulla riconfigurazione dinamica hanno evidenziato che il problema più significativo è capire l'istante temporale in cui effettuare l'aggiornamento di un nodo senza impattare sulla consistenza del sistema. Una policy di riconfigurazione dinamica stabilisce quindi quando e come effettuare l'update: i due approcci più validi nello stato dell'arte sono *Quiescence* e *Tranquillity*.
La Quiescence è un approccio sicuro e globale sull'intero sistema, e permette l'aggiornamento di un nodo solo dopo che abbia raggiunto lo stato di Quiescent. Questo particolare stato viene raggiunto bloccando qualsiasi futura computazione dei nodi vicini: in questo modo non si impatta negativamente sulla corretta esecuzione del sistema. Tuttavia introduce troppe latenze sul tempo di aggiornamento (*timeliness*) e sul tempo di interruzione del servizio (*disruption*).
La *Tranquillity* estende il criterio di Quiescence riducendo timeliness e disruption dell'aggiornamento. Questo approccio tuttavia non garantisce sempre un aggiornamento corretto nel caso di transazioni distribuite su più nodi del sistema.

L'approccio di *Version Consistency* raccoglie i pregi di entrambe le soluzioni garantendo riconfigurazioni sicure, che assicurano la correttezza globale del sistema con bassi tempi di aggiornamento e interruzione del servizio. Inoltre questo criterio è localmente verificabile, ovvero l'algoritmo che ne gestisce la logica è eseguito localmente sui nodi e non ha la necessità di avere un'entità centrale comune a tutti i componenti.
La Version Consistency aggiunge delle dipendenze dinamiche a quelle statiche definite nella configurazione architetturale: questi archi sono di due tipi, *future* per indicare un'interazione futura tra due nodi, e *past* per indicare una interazione avvenuta nel passato. Si dice configurazione *valida* una configurazione statica decorata con questi archi dinamici, e che soddisfa dei vincoli locali tra un componente e i componenti vicini.
Questi vincoli sono gestiti dall'algoritmo di *Gestione delle Dipendenze Dinamiche*, che permette ad un CBDS di mantenere attivamente una configurazione valida durante l'esecuzione delle transazioni.
L'algoritmo si può suddividere in tre parti: *Set-up*, *Progress* e *Clean-up*. La prima fase, quella di Set-up, è una fase di inizializzazione, effettuata quindi prima dell'esecuzione di transazioni, in cui un nodo crea degli archi dinamici di tipo future verso i componenti vicini ai quali richiederà servizio. La fase di Progress è effettuata invece a transazioni in corso: gli archi future sono rimossi gradualmente appena l'algoritmo è a conoscenza che il nodo non si servirà più di un altro,

e gli archi past sono creati per registrare che l'interazione è finita. Infine, nella fase di Clean-up, effettuata a transazioni concluse, vengono rimossi tutti gli archi past che sono stati creati precedentemente.
Nella Version Consistency la condizione che indica se un nodo è aggiornabile o meno si dice *freeness*: un nodo è free se non esiste una coppia di archi future/past entranti nel nodo che deve essere aggiornato.
Esistono tre strategie per il suo raggiungimento: *Waiting for freeness*, in cui il sistema aspetta che la freeness si manifesti spontaneamente sul nodo prima di effettuare l'aggiornamento; *Concurrent Versions*, che prevede la temporanea coesistenza di più versioni dello stesso componente per servire diversamente più transazioni e permettere aggiornamenti senza attese; *Blocking for Freeness*, che blocca il sistema evitando nuove transazioni sui nodi che interagiscono con quello da aggiornare, per raggiungere anticipatamente la condizione di free pur introducendo latenze dovute a interruzione del servizio.

Un'ampia fetta dei CBDS è formata da tutti quei sistemi che presentano una struttura orientata ai servizi: il *Software as a Service* e in particolare l'utilizzo dei *Web Services*, caratterizza le architetture formate da componenti eterogenei e dislocati su varie aree geografiche. Per realizzare sistemi di questo tipo è necessario però essere in grado di coordinare ed integrare i Web Services utilizzati dal sistema: *BPEL* è un linguaggio che permette proprio questo, ovvero di descrivere dei processi di business che coordinano i vari servizi del CBDS.
Un processo di business BPEL è formato da una serie di attività che vengono eseguite da più componenti web del sistema; il processo quindi corrisponde ad una transazione distribuita tra più nodi di un CBDS. Le attività più comuni di un processo BPEL sono quelle di invocazione (sincrona) verso un processo, ricezione e risposta ad un processo (asincrone). Ogni nodo del sistema distribuito è costituito da un unico processo BPEL che avrà il compito di interagire con altri processi BPEL collegati, allo scopo di effettuare operazioni di business e garantire il completamento della transazione distribuita generale.
Con l'implementazione della nostra tesi abbiamo sviluppato un framework per rendere sicuri e ottimali gli aggiornamenti dinamici di CBDS basati su processi di business, implementando il criterio di Version Consistency su una piattaforma reale.

Il framework che gestisce la Version Consistency è realizzato tramite due elementi principali:

1. Dynamo, una tecnologia che estende i servizi offerti dall'engine di esecuzione dei processi Active-Bpel, arricchendolo in particolare di una classe chiamata Interceptor. Questa classe permette di intercettare i processi BPEL in esecuzione sull'engine prima, durante e dopo il completamento di certe activity. Nei punti di intercettazione viene invocato il

2. VersionConsistencyService, che è un Web Service contenente la logica di business dell'algoritmo di riconfigurazione. Questo servizio ha al suo interno la struttura dati che modella gli archi dinamici e i metodi necessari per modificarla. Il Web Service svolge dunque anche il ruolo di database per il salvataggio della configurazione dinamica, per questo motivo è di tipo stateful, mantenendo in memoria lo stato del componente.

Ogni nodo del sistema distribuito è composto quindi da un processo BPEL che esegue una parte dell'intera transazione distribuita, da una istanza di Dynamo che esegue il processo e lo intercetta, e da un VersionConsistencyService che esegue la logica aggiungendo o rimuovendo gli archi. I nodi del sistema possono comunicare tra di loro attraverso chiamate SOAP tra i VersionConsistencyService o tra l'interceptor e un VersionConsistencyService; per mantenere la località dell'algoritmo queste interazioni possono avvenire solo tra componenti staticamente vicini tra di loro.
La configurazione statica è mappata attraverso stringhe presenti sia nell'interceptor che nel VersionConsistencyService, mentre la configurazione dinamica è modellata attraverso una sottoclasse presente nel VersionConsistencyService.
Le tre fasi dell'algoritmo di Version Consistency vengono eseguite in precisi punti dell'interceptor: la fase di Set-up e la fase di Clean-up prima e dopo l'esecuzione della transazione radice, nel nostro caso prima che parta Portal.bpel e appena dopo che ha terminato la sua esecuzione; la fase di Progress invece è divisa su tutti i punti di intercettazione toccati dall'esecuzione delle transazioni sui vari nodi.

La nostra implementazione permette la gestione dell'algoritmo anche se si eseguono più transazioni in concorrenza sul sistema: ogni arco è identificato dalla transazione root a cui appartiene e il suo ciclo di vita è gestito in modo totalmente indipendente dagli archi appartenenti alle altre transazioni.

L'aggiornamento di un componente è stato implementato modificando dinamicamente l'endpoint reference del servizio di business orchestrato da un determinato processo BPEL: l'aggiornamento viene realizzato tramite la sostituzione del web service che ne implementa la logica di business.
Il criterio di raggiungimento della freeness che abbiamo sfruttato è quello più performante di *Concurrent Versions*, che permette di ridirezionare le transazioni rispetto alle quali un nodo è free sulla nuova versione, mantenendo l'esecuzione delle altre sulla vecchia.

La tesi è dunque organizzata nel seguente modo: il capitolo 2 contiene lo stato dell'arte, descrive i criteri di Quiescence e Tranquillity, e propone la formalizzazione di CDBS e della sua riconfigurazione dinamica. Il capitolo 3 descrive nel dettaglio il criterio di Version Consistency, evidenziandone i pregi rispetto alle soluzioni presentate nel capitolo 2. Il capitolo 4, dopo una breve introduzione sui

processi di business, mostra la nostra implementazione della Version Consistency calata in questo contesto. Il capitolo 5 evidenzia la validità della nostra soluzione, mostrando le simulazioni effettuate e motivando alcune scelte implementative fondamentali. Infine, nel capitolo 6 presentiamo le conclusioni di questo lavoro di tesi e i possibili sviluppi futuri.

Capitolo 2

Il problema della riconfigurazione dinamica: lo stato dell'arte

2.1 Aspetti generali

Negli ultimi anni nel settore informatico si è verificata la rapida diffusione di un modello di sistemi radicalmente diverso rispetto al passato.
L'utilizzo dei grandi sistemi centralizzati sta calando di fronte all'enorme crescita dei sistemi distribuiti, ovvero architetture informatiche composte da elementi eterogenei e geograficamente separati tra loro: diversi componenti hardware e software vengono integrati e "composti" per raggiungere gli obiettivi funzionali del sistema.
Non esiste un unico modello di sistema distribuito ma solitamente il concetto di distribuzione è garantito dalla presenza di diversi componenti che vengono integrati tra loro. Proprio per questo motivo viene utilizzato l'acronimo ***Component Based Distributed Systems*** o più semplicemente la sigla ***CBDS***.
Un sistema distribuito integra quindi diverse piattaforme hardware e software e può essere presente su più aree geografiche, anche a livello mondiale.
Sono di nostro particolare interesse i sistemi basati sulla tecnologia Web, che possono vedere coinvolte reti di diverso tipo (Internet, Intranet, reti Mobile), la cui realizzazione è stata resa possibile da alcuni concetti che verranno presentati nei capitoli successivi come *l'architettura SOA,* l'idea di *Software as a Service (SaaS)*, la *"composizione" di Web Services*.
Lo sviluppo continuo del Web, la diffusione di strumenti che garantiscono interoperabilità e facilità di comunicazione come XML o WSDL e in parte anche l'ascesa del linguaggio Java sono tutti elementi che si sono rivelati decisivi per l'affermazione di questo nuovo paradigma software.

Un CBDS è per definizione un sistema eterogeneo caratterizzato dalla presenza di macchine, linguaggi e strutture dati differenti; di conseguenza sarà molto

difficile integrare e gestire la cooperazione tra i vari componenti.
Tuttavia tecnologie come Java, XML, WSDL (come accennato in precedenza) facilitano il compito permettendo di gestire la comunicazione tra componenti con maggior facilità e consentono ai CBDS di essere una soluzione architetturale moderna ed efficace.
Ovviamente un sistema distribuito dovrà anche essere sicuro, garantendo l'integrità dei dati e dei permessi di accesso, ed essere scalabile, evitando crolli o rallentamenti.

Uno dei problemi legati ai CBDS che ha suscitato maggior interesse è però l'*aggiornamento* dei singoli componenti del sistema, che in caso di necessità deve avvenire il più velocemente possibile ma in modo totalmente trasparente e sicuro, garantendo cioè l'integrità dell'intero sistema.
I CBDS devono infatti fronteggiare dei cambiamenti nei requisiti che devono essere soddisfatti in tempi rapidi: è infatti impensabile che nell'epoca attuale un sistema software, anche se distribuito, resista per anni senza essere mai aggiornato o senza subire cambiamenti.
Nel caso dei CBDS la gestione degli aggiornamenti diventa ancora più complessa perché bisogna spesso intervenire su un singolo componente, evitando però di impattare in maniera pesante sulle funzionalità e sulla logica di business dell'intero sistema: a volte è infatti necessario modificare parti esistenti dell'implementazione o aggiungere nuove funzionalità a runtime, senza la possibilità di bloccare il sistema.
Rispetto alle modifiche off-line, gli *aggiornamenti dinamici* sono decisamente più complessi poiché bisogna garantire la correttezza delle attività in corso (anche se si agisce su un singolo componente, è sempre la stabilità dell'intero sistema ad essere messa in gioco); in più è fondamentale minimizzare l'interruzione del servizio e il ritardo con cui il componente viene aggiornato.
Come è possibile gestire tutto ciò?

2.2 Il problema della riconfigurazione dinamica

Il problema della riconfigurazione dinamica (aggiornamenti a runtime) è un concetto chiave per i sistemi distribuiti basati su componenti (CBDSs), che devono essere progettati in modo da reagire ai cambiamenti dell'ambiente in cui sono calati e dei requisiti che devono soddisfare.
Questi cambiamenti possono essere difficili da predirre a design time o troppo costosi da identificare e gestire dal software inizialmente progettato. É così necessario modificare le parti esistenti dell'implementazione o aggiungere nuove funzionalità a runtime, senza cioè fermare il sistema.

I sistemi ditribuiti vengono usati ad esempio per gestire servizi e-mail, motori di ricerca, giochi online, sistemi di processamento di dati scientifici o finanzia-

ri, reti di distribuzione di contenuti multimediali, gestione di file sharing e così via: per questo i sistemi che gestiscono tali servizi avranno frequenti necessità di cambiamenti (risoluzioni di bugs, aggiunta di funzionalità, miglioramento delle performance, ecc.); ovviamente, mentre il sistema generale viene aggiornato, sarà comunque necessario continuare a garantire il servizio agli utenti, questo perchè i periodi di downtime sono sempre meno tollerati, basti pensare a possibili danni economici causati dall'interruzione di un sistema di vendite on-line, o di un servizio di assistenza medica remota che potrebbe addirittura mettere in gioco vite umane.

Un ambito di ricerca molto interessante che si è sviluppato recentemente nell'ingegneria del software è quindi l'aggiornamento dinamico dei sistemi distribuiti. La riconfigurazione dinamica dei CBDSs non è banale: comparate con la manutenzione offline, le modifiche dinamiche sono più difficoltose poichè, oltre alla correttezza della nuova versione, devono garantire il corretto completamento delle attività (*transazioni*) che il sistema sta eseguendo.
Allo stesso tempo si deve minimizzare il tempo di interruzione del servizio (chiamato *disruption*) e il ritardo durante il quale avviene l'aggiornamento (chiamato *timeliness*).[10]

Un tema comune di molti articoli a riguardo pubblicati in letteratura è quindi la scelta del *momento giusto* in cui effettuare l'aggiornamento: l'istante di tempo in cui sostituire la vecchia versione di un componente con la versione aggiornata è un punto fondamentale del problema.
L'obiettivo è il raggiungimento di un nuovo stato valido del sistema da cui si dovrà partire per continuare a garantirne le funzionalità. Tutto ciò deve avvenire in modo *safe*, cioè senza perdite di dati e/o modifiche alle funzionalità stesse, e possibilmente in maniera automatica, visto che per alcuni sistemi di notevoli dimensioni è impensabile che una persona possa fisicamente stare ad aspettare il momento giusto per aggiornare un componente. Questa è la vera sfida della riconfigurazione nei CBDS.

Molti studiosi hanno proposto delle soluzioni al problema nei loro articoli di ricerca: invece di passare subito da una vecchia ad una nuova versione post-aggiornamento, alcuni lavori si focalizzano sul mantenimento di una versione intermedia temporanea per facilitare l'aggiornamento del sistema.
Ad esempio Zhang e Cheng [12] proposero un approccio basato su modelli per lo sviluppo di software adattabili a runtime: i comportamenti delle differenti versioni di un'applicazione venivano modellati attraverso appositi diagrammi a stati come ad esempio le reti di Petri (che rappresentavano appunto i modelli formali utilizzati per lo studio del problema). I cambiamenti di versione venivano modellizzati come transazioni e cambiamenti di stato, ed agendo sui diagrammi si cercava di capire quando aggiornare il sistema in modo sicuro.
Biyani e Kulkarni [2] modellizzarono invece il problema attraverso dei reticoli

transazionali. Con questo approccio, basato anch'esso sui modelli, erano in grado di studiare le sequenze di transazioni corrette per garantire un sistema adattivo che permettesse gli aggiornamenti da vecchie a nuove versioni del software.

Il punto cruciale per quanto riguarda gli aggiornamenti dei CBDS è l'identificazione di condizioni adatte all'intervento sui singoli componenti del sistema.
I due approcci principali proposti in letteratura per la risoluzione di questo problema sono il metodo basato sulla *quiescence* proposto da Kramer e Magee [7] e il metodo basato sulla *tranquillity* sviluppato da Vandewoude [11].
Un esempio di approccio generale e sicuro è quello che permette l'aggiornamento dei componenti dopo che essi abbiano raggiunto lo stato di *quiescence*. Questo criterio (che vedremo più in dettaglio nel prossimo paragafo) ha il vantaggio di separare nettamente la computazione e la (ri)configurazione archietturale: esso blocca qualsiasi computazione dipendente dal nodo da aggiornare prima di abilitare qualsiasi tipo di aggiornamento; il problema è che questo criterio introduce troppa disruption rispetto a quella realmente necessaria per aggiornare il componente stesso.
L'approccio basato su quiescence considera solamente le dipendenze *statiche* tra i componenti, specificate dalla configurazione architetturale; siccome queste dipendenze includono tutte le possibili interazioni tra il nodo da aggiornare e i suoi vicini, è possibile ridurre il tempo di disruption considerando invece le dipendenze *dinamiche*. Esse sono relazioni temporali tra i componenti causate da transazioni in corso, e quindi mettono in evidenza solamente i vincoli *correnti* sulla riconfigurabilità del sistema. L'approccio presentato e implementato in questa tesi le sfrutta per assicurare la consistenza "globale" di transazioni multi-distribuite attraverso la nozione di *version consistency*. Come già detto, le riconfigurazioni dinamiche sono viste come sequenze di aggiornamenti effettuati a runtime; la version consistency assicura che ogni transazione sia servita dalla vecchia versione o dalle nuove, indipendentemente dall'istante in cui avviene l'aggiornamento.
Questo approccio introduce meno disruption rispetto a quello basato su quiescence, infatti ogni componente può essere aggiornato anche se non ha ancora raggiunto lo stato di quiescence, l'importante è che non sia stato ancora usato né verrà usato più dalle transazioni in corso sul nodo interessato.

Bidan, Issarny, Saridakis e Zarras [3] invece studiarono un sistema per l'aggiornamento dinamico dei sistemi CORBA che risultò interessante anche ad un livello più generale: essi svilupparono un algoritmo innovativo che permetteva di portare i nodi affetti dalla riconfigurazione dinamica in uno stato di passività. Le attività dei nodi passivi venivano bloccate senza però bloccare interamente l'operato dell'intero sistema, che proseguiva con l'esecuzione delle transazioni che non subivano l'impatto degli aggiornamenti.
Chen e Simons [4] cercarono si sviluppare un framework che offrisse meccanismi per analizzare le interazioni tra i componenti di un CBDS durante una riconfi-

gurazione dinamica, in modo da favorire la comunicazione tra di essi per evitare aggiornamenti che causassero danni al sistema.

Una teoria favoreggiata da molti ricercatori e studiosi afferma che l'aggiornamento dinamico a runtime dei sistemi software dovrebbe essere modellizzato, analizzato e gestito a livello architetturale.
I modelli architetturali forniscono una visione astratta ma globale del sistema e specificano i vincoli di integrità che devono essere preservati dalla riconfigurazione. Ma questi modelli falliscono nel momento in cui bisogna individuare le dipendenze dinamiche a runtime relative alle varie transazioni in corso sul sistema. Come notato prima, non è possibile assicurare un aggiornamento in condizioni di sicurezza analizzando semplicemente la configurazione statica, ma è necessario fronteggiare anche i problemi relativi alla configurazione dinamica.

Taentzer, Goedicke e Meyer [5] cercarono di proporre una soluzione che gestisse la sicurezza dell'aggiornamento effettuando considerazioni unicamente sugli aspetti strutturali statici, prescindendo invece dagli aspetti dinamici. Affermavano che non ci fosse bisogno di considerare lo stato presente del sistema, ma in realtà poi fu evidente che da esso non si può prescindere per un update corretto. Wermelinger, Lopes e Fiadeiro [8] presentarono un linguaggio di alto livello per descrivere le architetture e per gestirne gli aggiornamenti; però i modelli formali che non tengono conto dello stato dinamico del sistema non sono sufficienti.

L'approccio migliore per l'aggiornamento dei CBDS su cui si basa anche l'intero lavoro del nostro progetto di tesi è il criterio di *version consistency*, sviluppato in precedenza da Neamtiu [6] e successivamente nell'articolo di Ma, Baresi, Ghezzi, La Manna e Lu [10]. Essendo alla base della nostra implementazione, presenteremo molto dettagliatamente questo criterio nel capitolo 3 ma prima discuteremo di *quiescence* e *tranquillity* in modo da poter eseguire un successivo confronto con la *version consistency*.

2.3 Definizioni formali

Un sistema distribuito component-based può essere descritto come un insieme di componenti (*nodi*) con porte in ingresso/uscita. I nodi sono collegati da archi orientati dalle porte in ingresso a quelle in uscita. Il grafo risultante è chiamato *configurazione statica* del sistema, nella quale ogni nodo è identificato con la versione corrente del componente che rappresenta.

2.3.1 Configurazione statica

La configurazione statica di un sistema distribuito basato su componenti è un grafo ordinato i cui nodi rappresentano i componenti con porte ingresso/uscita.

Un arco orientato, che collega la porta in uscita di un nodo N alla porta in ingresso di un altro nodo M, rappresenta una dipendenza statica (ad esempio, la possibilità che N possa richiedere un servizio da M).

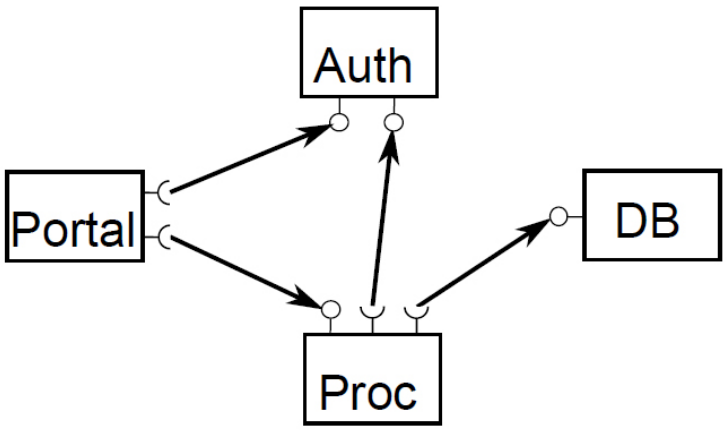


Figura 2.1: Esempio di configurazione statica.

La figura 2.1 mostra un sistema di esempio molto simile a quello utilizzato nel corso della tesi. Un componente funge da portale per gli utenti, **Portal**, il quale interagisce con un componente di autenticazione (**Auth**) e un processo di gestione business (**Proc**); Proc a sua volta interagisce sia con Auth che con un componente database (**DB**). Questo significa che Portal staticamente dipende da Proc e Auth, mentre Proc dipende da Auth e DB.

2.3.2 Transazione

Un componente può eseguire transazioni. Una *transazione* è una sequenza di azioni eseguite da un componente, completate in un certo lasso di tempo. Le azioni includono computazioni locali e scambi di messaggi.
Le transazioni sono rappresentate dai rettangoli $T_0 ... T_4$ nel sequence diagram mostrato in figura 2.2. Esso rappresenta un tipico caso d'uso del nostro esempio: una transazione T può essere iniziata da un client esterno o da un'altra transazione T': nel primo caso T è chiamata *transazione root (root transaction)* mentre nel secondo *sotto-transazione (sub-transaction) di* T'.
Se una transazione T inizia una sottotransazione, l'insieme formato da T e da tutte le sue sottotransazioni è detto *transazione distribuita*. Per esempio, T_4 è la sottotransazione di T_2 nel sequence diagram di figura 2.2 .
Il termine *sub*(T_1, T_2) denota che T_2 è una sottotransazione *diretta* di T_1: una transazione può essere sottotransazione diretta di una sola transazione.
L'insieme $ext(T) = \{x | x = T \vee sub^+(T, x)\}$ è l'*insieme esteso (extended transac-*

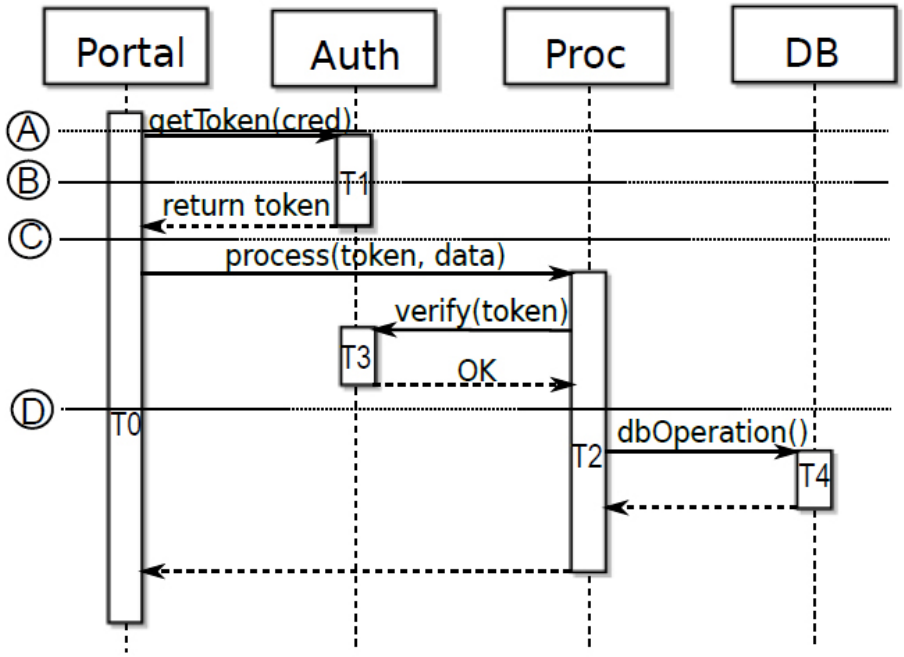


Figura 2.2: Scenario di esempio.

tion set) della transazione T, il quale contiene T e tutte le sue dirette e indirette sottotransazioni; l'insieme esteso di una transazione modella il concetto di *transazione distribuita* su più componenti.
Una transazione eseguita su un componente può iniziare una sottotransazione su un componente vicino solo quando quest'ultimo è dipendente staticamente dal primo. Riprendendo l'esempio, T_0 inizia T_1 e T_2; successivamente T_2 inizia T_3 e T_4. Nel nostro modello assumiamo sempre che le transazioni sono sincronizzate tra di loro, nel senso che ricevono la notifica del completamento di una loro sottotransazione e non possono terminare prima di essa.

Riprendendo il nostro caso d'uso di figura 2.2: Portal riceve un token di autenticazione da Auth e lo usa per richiedere il servizio da Proc; Proc verifica il token attraverso Auth e comincia la propria computazione interagendo con il database DB. Se consideriamo la transazione root T_0 su Portal, il suo extended transaction set è dato da $ext(T_0) = \{T_0, T_1, T_2, T_3, T_4\}$, dove T_1 su Auth risponde alla richiesta getToken, T_2 su Proc risponde alla richiesta process, T_3 su Auth risponde alla richista di verifica del token verify e T_4 alle richieste a operazioni sul database effettuate da T_2.

2.3.3 Aggiornamenti a runtime

Una policy di riconfigurazione dinamica definisce *quando* e *come* effettuare l'aggiornamento per garantire la consistenza del sistema.
Una riconfigurazione dinamica è una sequenza atomica di runtime updates di componenti di un sistema distribuito. Più rigorosamente, un update è specificato dalla tupla $< \sum, \omega, \omega', \Gamma, s >$, dove $\sum$ è la configurazione originale del sistema e ω è il set dei componenti che devono essere sostituiti dalla nuova versione ω'. L'aggiornamento avviene quando il sistema si trova nello stato s e la funzione di trasformazione dello stato Γ trasforma s in $s' = \Gamma(s)$, che è lo stato del sistema con configurazione $\sum' = \sum[\omega/\omega']$. Il sistema continuerà l'esecuzione da s' senza errori visibili.

Assumiamo che per un dato aggiornamento a runtime $< \sum, \omega, \omega', \Gamma, s >$ il corrispondente aggiornamento off-line $< \sum, \omega, \omega' >$ sia corretto. La *correttezza dell'aggiornamento off-line* sta a significare che le transazioni che si stanno eseguendo su $\sum$ soddisfano le vecchie specifiche del sistema mentre le transazioni su $\sum' = \sum[\omega/\omega']$ soddisfano le nuove. Data la natura distribuita di queste transazioni possiamo dare solo una definizione sommaria di *correttezza di aggiornamento a run-time*:

- le transazioni che finiscono prima dell'update soddisfano le vecchie specifiche;
- le transazioni che cominciano dopo l'update soddisfano le nuove specifiche;
- le transazioni che cominciano prima e finiscono dopo l'update soddisfano o le vecchie o le nuove specifiche.

Per esempio, se consideriamo il caso visto prima, si può ipotizzare di aggiornare Auth per avere funzioni crittografiche più efficienti e così migliorare la sicurezza dell'intero sistema. Nonostante le nuove funzioni crittografiche risultino incompatibili con le vecchie, gli altri componenti del sistema non devono essere aggiornati perchè tutte le operazioni di encryption/decryption sono effettuate dal solo Auth. Le specifiche degli altri componenti rimangono così inalterate e un aggiornamento fatto offline sarebbe molto semplice da effettuare; il problema è che se aggiorniamo Auth a runtime, dobbiamo anche assicurare che tutte le transazioni correnti possano essere eseguite correttamente prima e dopo l'update.
Se la policy di riconfigurazione permette l'aggiornamento in ogni istante di tempo, è molto difficile assicurare questo tipo di correttezza; una policy che assicura *quando* aggiornare in modo safe è il verificarsi della condizione di *inattività*.

Un componente è ***inattivo (idle)*** *se e solo se non sta eseguendo transazioni e il suo stato locale è equivalente allo stato iniziale.*

L'assunzione che i componenti possano essere aggiornati solo se inattivi è insufficiente per garantire aggiornamenti sicuri: infatti se consideriamo lo scenario di figura 2.2, e sostituiamo Auth quando inattivo ma deve servire ancora la richiesta verifyToken (cioè in concomitanza dell'istante temporale (C)), il sistema risultante non sarà corretto, poichè il token di sicurezza sarà creato con un algoritmo (vecchia versione di Auth) e validato con un altro (nuova versione di Auth).

In generale, una policy di riconfigurazione dinamica deve assicurare le seguenti condizioni:

1. forte abbastanza per assicurare la correttezza degli aggiornamenti;
2. debole abbastanza per assicurare bassa disruption;
3. automaticamente verificabile in un ambiente distribuito (localmente sui nodi).

2.4 Gli approcci più comuni in letteratura

2.4.1 Quiescence

In un paper scientifico Kramer e Magee [7] hanno proposto un criterio di riconfigurazione chiamato *quiescence*, secondo loro condizione sufficiente per manipolare un nodo in modo sicuro durante un processo di aggiornamento dinamico. Essi furono anche i primi a modellare un sistema distribuito con un grafo orientato.

Nel loro articolo definirono la transazione come *"scambio di informazioni tra due (e solo due) nodi connessi, cominciato da uno dei due nodi. Le transazioni sono il motivo per cui lo stato di un singolo nodo è affetto dagli altri nodi vicini connessi ad esso nel sistema. Le transazioni consistono di una sequenza di uno o più scambi di messaggi tra due nodi connessi. Si assume che le transazioni vengono completate in un lasso di tempo finito e che il nodo che comincia (che inizia) l'esecuzione di una transazione deve essere al corrente del suo completamento".*

Una transazione T può dipendere da altre transazioni T_i: il completamento di T dipende dal completamento di tutte le transazioni T_i. Questo definizione di transazione differisce da quella fornita da noi nei paragrafi precedenti: corrisponde alla definizione solo nel caso di transazioni che interessano solamente due nodi. Tuttavia attraverso la nozione di dipendenze dinamiche, Kramer e Magee inglobarono il concetto di transazione distribuita tra nodi multipli.
Kramer e Magee descrivono lo stato di un sistema come un insieme di stati di configurazione per ogni nodo e tra di essi considerano due stati principali, attivo e passivo, dei quali riportiamo le definizioni:

- ***Stato Attivo.*** *Un nodo nello stato di Attivo inizia, accetta e serve transazioni.*

- ***Stato Passivo.*** *Un nodo nello stato di Passivo deve continuare ad accettare e servire transazioni ma:*

  1. *non è occupato in una transazione che ha iniziato in passato e*
  2. *non inizierà nuove transazioni nel futuro.*

Kramer e Magee specificano che lo stato passivo è condizione necessaria ma non sufficiente per l'aggiornamento, infatti un nodo, raggiunto questo stato, potrebbe comunque stare eseguendo transazioni iniziate da altri nodi connessi staticamente ad esso. Per questo motivo introdussero una condizione più forte, la condizione di *Quiescence*: *Un nodo è* ***quiescent*** *se:*

1. *non è coinvolto in una transazione che lui stesso ha iniziato;*
2. *non inizierà transazioni nel futuro;*
3. *non sta servendo transazioni;*
4. *nessuna transazione è stata o sarà iniziata da un altro nodo che richiede il servizio di questo nodo.*

Come detto, un nodo che soddisfa le prime due condizioni è detto *passivo*; le ultime due condizioni invece rendono il nodo indipendente da tutte le transazioni esistenti o che verranno eseguite in futuro, e così può essere aggiornato in modo *safe*; per portare un nodo allo stato di quiescence, oltre alla passivazione, è necessario che tutti i nodi staticamente dipendenti da esso siano anch'essi passivati per assicurare le ultime due condizioni.

L'algoritmo di gestione della quiescence si chiama *Change Managemen Protocol*, che prevede dei cambiamenti di stato sui nodi attraverso azioni strutturali, *create, remove, link, unlink, activate, passivate* (figura 2.3).
Il protocollo cerca di stabilire una cosiddetta *regione* di quiescence, che rappre-

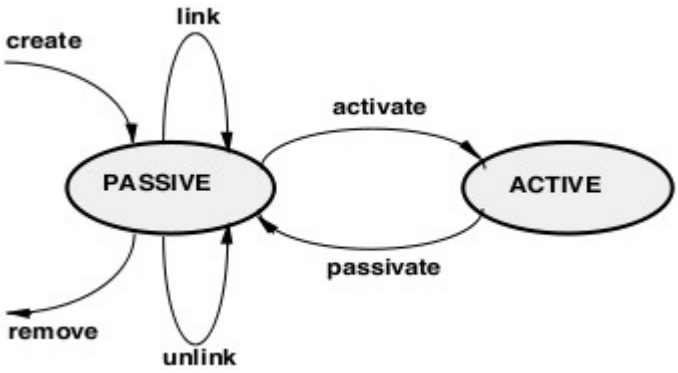


Figura 2.3: Stati e azioni di un nodo nel protocollo di change.

senta l'insieme di nodi che devono raggiungere lo stato passivo per permettere l'update di un determinato nodo. Questo viene effettuato tramite le azioni di cambiamento di figura 2.3, secondo le seguenti **regole**:

- **Rimozione dei nodi - remove.** La precondizione per rimuovere un nodo N è che esso deve essere *isolato*, cioè non deve avere connessioni dirette con altri nodi: un nodo isolato non interferisce sul sistema e quindi può essere rimosso.

- **Connessione dei nodi - link e unlink.** La precondizione per collegare o scollegare un nodo N è che esso deve essere nello stato di quiescent: in questo modo lo stato del nodo N è consistente e il collegamento/scollegamento avviene in un contesto stabile e sicuro.

- **Creazione di un nodo - create.** La precondizione è ovviamente sempre vera: quando un nodo è creato, è inizialmente isolato e conseguentemente deve essere per forza nello stato di quiescent siccome non può né rispondere né iniziare transazioni.

- **Attivazione/passivazione di un nodo - activate/passivate.** Non ci sono regole teoriche per attivare/passivare un nodo, dipende dall'implementazione del device su cui è installato l'algoritmo.

Una sequenza di azioni che deriva dai cambiamenti strutturali dei nodi da aggiornare, per soddisfare le pre-condizioni evidenziate qui sopra, si chiama *change transactions*. Un possibile algoritmo che sfrutta le change transaction agisce in questo modo:

1. **step 1**: determina l'insieme di connessioni CS che devono essere rimosse per isolare i nodi da rimuovere (e soddisfare la prima precondizione). Da qui, tramite l'insieme alle connessioni LS direttamente specificate dalle direttive link o unlink, si determina l'insieme di nodi QS (Quiescent Set) che devono essere quiescent per soddisfare la seconda precondizione;

2. **step 2**: forma l'insieme CPS (change passive set) come unione degli insiemi PS per ogni ogni nodo presente in QS;

3. **step 3**: esegui le azioni di configurazione nel seguente ordine: passivate (di tutti i nodi in CPS) - unlink - remove - create - link - activate (per tutti i nodi di CPS togliendo i nodi rimossi e i nodi creati).

Il risultato di queste operazioni è il sistema aggiornato con la sostituzione dei nodi target dell'update.

Sebbene la quiescence risulti essere una condizione sufficiente per l'aggiornabilità, ha il problema che introduce disruption troppo alta nel sistema in esecuzione,

infatti non solo il nodo da aggiornare deve raggiungere lo stato di passività, ma anche tutti i nodi direttamente o indirettamente capaci di iniziare transazioni su questo nodo. Il criterio che descriviamo di seguito cerca di risolvere questo problema.

2.4.2 Tranquillity

Vandewoulde nel paper *Tranquillity: a low disruptive alternative to Quiescence for ensuring safe dynamic updates* [11], espone un altro approccio per la riconfigurazione dinamica dei CDBS, tentando di limitare il tempo di interruzione di servizio, troppo alto nella soluzione proposta da Kramer e Magee.

Il criterio di Tranquillity è basato su due osservazioni:

- non c'è nessun problema nel rimpiazzare un nodo mentre una transazione è attiva, se il nodo sostituito non è coinvolto in quella transazione. Questo significa che un nodo che ha partecipato in una transazione in esecuzione potrebbe essere sostituito se si è certi che il nodo non parteciperà più in quella transazione. É anche permesso l'update di un nodo se potrà partecipare a una transazione in esecuzione, ma non ha ancora partecipato ad essa. Questo concetto è riassunto nell'immagine 2.4.

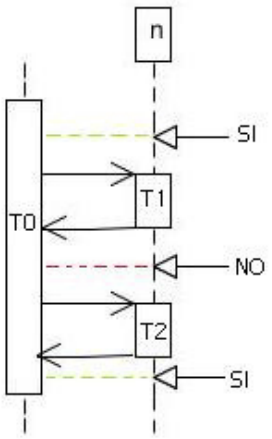


Figura 2.4: Tranquillity applicata ad un generico nodo n.

- Usare un design simil black-box per i nodi di un CDBS è un buon approccio per favorire la riusabilità e disaccoppiare le parti di un sistema. Questo implica che i nodi possono richiedere servizio ad altri nodi ai quali sono connessi, senza però sapere nulla della loro implementazione. Se tutti i nodi sono delle black-box, allora tutti i partecipanti di una transazione

rappresentano o l'iniziatore o sono direttamente connessi ad esso. I nodi indirettamente connessi all'iniziatore di una transazione possono anche non partecipare a una transazione eseguita dall'iniziatore siccome la loro esistenza è sconosciuta ad esso.

Questi due concetti sono inglobati nella definizione di *Tranquillity: un nodo è* ***tranquillo*** *se:*

1. *non è coinvolto in una transazione che lui stesso ha iniziato;*
2. *non inizierà transazioni nel futuro;*
3. *non sta attivamente processando richieste;*
4. *nessuno dei nodi adiacenti è coinvolto in una transazione nella quale esso ha sia già partecipato, sia potrebbe partecipare nel futuro.*

La Quiescence è una condizione più forte rispetto alla Tranquillity, nel senso che implica la Tranquillity ma non viceversa.
La Condizione 3 della Quiescence implica che il nodo da aggiornare non stia né processando attivamente richieste, né attendendo nuove richieste da altre transazioni attive; questo banalmente implica la condizione 3 della Tranquillity.
La condizione 4 della Quiescence puntualizza che nessun nodo vicino al nodo N ha iniziato o inizierà transazioni nei quali N partecipa; questo significa che nessuna transazione è attiva, banalmente includendo la condizione 4 della Tranquillity.
Viceversa, la Tranquillity non implica la Quiescence, infatti non richiede che i nodi connessi al nodo da aggiornare N potrebbero non iniziare transazoni che involvono N; per la Tranquillity non è necessario che che i nodi direttamente connessi ad N siano passivati; questo ha il vantaggio che è decisamente più performante, introducendo molta meno disruption.

Tuttavia, per stessa ammissione dell'autore del paper (come vedremo nel capitolo 3), la Tranquillity ha dei limiti. Quello più evidente è che lo stato di aggiornabilità di un componente potrebbe non essere mai raggiunto. Questo succede, ad esempio, nel caso in cui il componente è usato in una sequenza infinita di transazioni intrecciate tra di loro.
Un secondo svantaggio consiste nel fatto che il criterio di tranquillity non è stabile se usato da solo, infatti una volta che un nodo ha raggiunto la tranquillity, tutte le interazioni tra di esso e i nodi vicini devono essere bloccate per garantire lo stato durante l'aggiornamento vero e proprio. Questo non succede per il criterio di Quiescence, poichè l'aggiornamento avviene quando tutti i nodi coinvolti sono passivati, e successivamente riattivati esplicitamente dall'algoritmo di change.
Come vedremo nel capitolo 3, la Tranquillity calata nel nostro esempio mostra un altro grosso limite: non garantisce la consistenza degli aggiornamenti in particolari casi d'uso che prevedono transazioni distribuite particolarmente estese.

Capitolo 3

Algoritmo di Version Consistency

3.1 I limiti di Quiescence e Tranquillity

Prima di introdurre l'algoritmo utilizzato in questa tesi, è utile comprendere il motivo di questa scelta rispetto agli altri algoritmi/criteri di riconfigurazione dinamica presenti in letteratura. Abbiamo visto nel capitolo 2 che le due soluzioni più interessanti sono quelle di Quiescence e Tranquillity, ma queste presentano dei limiti evidenti in termini di tempo e sicurezza degli aggiornamenti.

La politica di Quiescence prevede che un nodo di un sistema distribuito sia aggiornabile solo nel momento in cui tutti i nodi che richiedono servizio ad esso raggiungono lo stato di passività. Se quindi si segue questo approccio per la riconfigurazione di un CBDS, un nodo non può essere quiescent prima del completamento di tutte le transazioni eseguite dai nodi staticamente dipendenti; ciò significa che l'aggiornamento può subire pesanti ritardi.
Se riprendiamo il nostro sistema di esempio, Auth non può essere quiescent prima che Portal e Proc terminino le loro transazioni, (rispettivamente T_0 e T_2) come si nota in figura 3.1.

In più, tutti gli altri nodi che possono potenzialmente eseguire transazioni che richiedono servizio ad Auth (direttamente o indirettamente) sono passivati, e restano bloccati fino alla fine dell'aggiornamento. Ancora, nel nostro esempio, Portal e Auth devono essere passivati prima dell'update di Auth: questo significa che l'adozione di questo criterio all'interno di un algoritmo di riconfigurazione può introdurre parecchia disruption al servizio offerto dal sistema.

Per ridurre la disruption, Vandewoulde [11] ha proposto il concetto di *tranquillity* come alternativa alla quiescence.
L'idea base è che, per qualsiasi nodo da aggiornare:

- non è più necessario aspettare la terminazione di una transazione dipendente, se la transazione *non richiederà più* servizio dal nodo (in figura 3.2

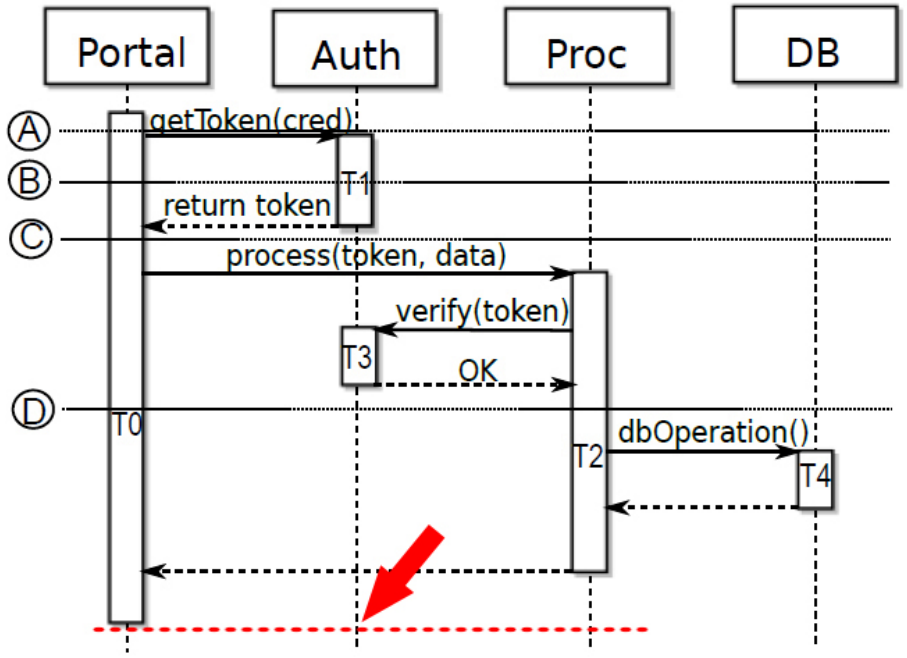


Figura 3.1: Auth raggiunge lo stato di quiescent solo all'istante temporale indicato in rosso.

è l'istante temporale dopo T_2);

- è permesso l'update anche se dipendente da transazioni di nodi adiacenti, se l'interazione *non è ancora avvenuta* (in figura 3.2 è l'istante temporale prima di $T1$).

Anche se nel paper di Vandewoulde si afferma che la tranquillity è una "condizione sufficiente per la consistenza delle applicazioni durante una riconfigurazione dinamica [11], la nozione di tranquillity è basata su una assunzione troppo forte: applicando la definizione al nostro modello, la nozione di transazione distribuita non può essere completamente corretta come definito precedentemente.
Infatti, sotto l'assunzione imposta dalla tranquillity, una transazione distribuita iniziata da una transazione root T sul nodo N può contenere solo sotto-transazioni eseguite dai nodi adiacenti direttamente connessi ad N: questo significa che una sotto-sotto-transazione (ad esempio una sottotransazione iniziata da un'altra sottotransazione) eseguita da un nodo non direttamente connesso ad N, non è parte della transazione distribuita e per questo motivo può essere aggiornato senza problemi, essendo un'entità totalmente indipendente.

Ma questa limitazione permetterebbe updates *non safe* nel nostro scenario di esempio, come si può notare in figura 3.3. Infatti, dopo che Auth ha ritornato il token a Portal, esso non verrà più chiamato direttamente dalla transazione di Portal T_0; in più, prima che arrivi la richiesta di verifica del token, Auth non è

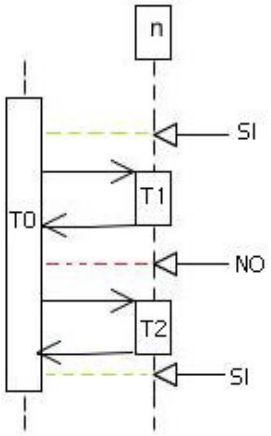


Figura 3.2: Tranquillity applicata ad un generico nodo n.

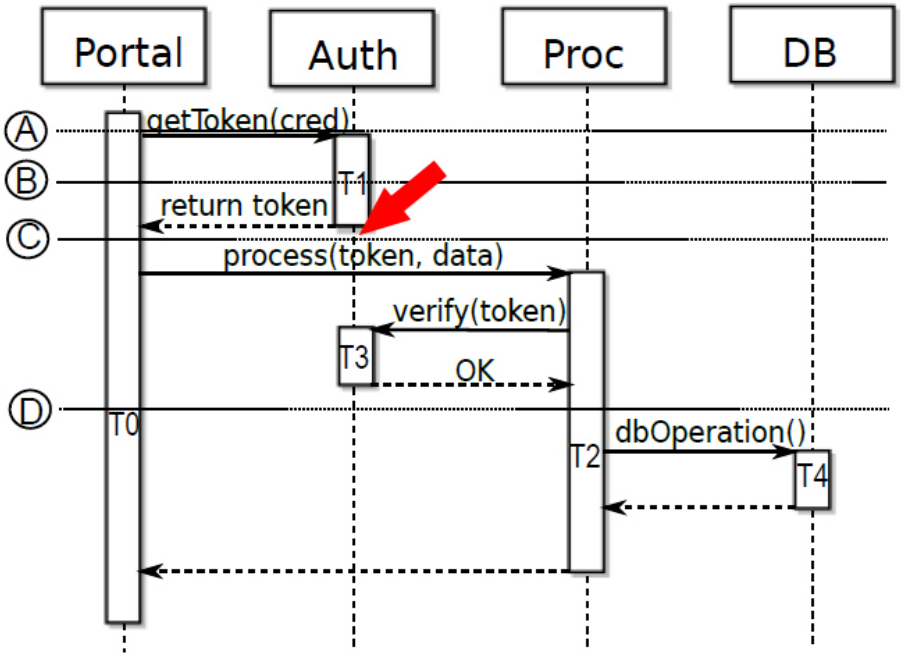


Figura 3.3: Auth raggiunge lo stato di tranquillity all'istante temporale indicato in rosso.

ancora parte della sessione iniziata da Proc, per cui al punto (C) il nodo Auth è *tranquillo*.
Tuttavia, se Auth fosse aggiornato in questo istante, la verifica del token fallirebbe poichè il token sarebbe stato generato dalla vecchia versione di Auth, il quale, dopo l'aggiornamento, potrebbe utilizzare un diverso algoritmo crittografico per validarlo. Questo errore non si presenterebbe se si eseguisse lo scenario interamente con la vecchia versione o interamente con la nuova.

Per concludere, possiamo affermare che l'approccio basato su quiescence è una soluzione safe e generale (mantiene la consistenza dell'intero sistema distribuito), ma può introdurre troppa disruption e tempi di aggiornamento lunghissimi. L'approccio basato su tranquillity invece è più performante, ma si basa su assunzioni che non ne permettono l'uso su sistemi troppo grandi (mantiene consistenza e safeness dell'aggiornamento solo locale sui nodi) che possono cioè eseguire una fitta "rete" di transazioni.
L'approccio di version consistency invece raccoglie i punti forti di entrambe le soluzioni, garantendo riconfigurazioni veloci e che mantegono sicurezza sull'intero CBDS; è per questo che abbiamo deciso di utilizzarla nella nostra tesi. Anticipiamo i vantaggi dell'implementazione della Version Consistency mostrando l'immagine 3.4, che mette in evidenza l'istante temporale in cui è possibile aggiornare Auth applicando questo criterio al nostro esempio. Come si nota in figura,

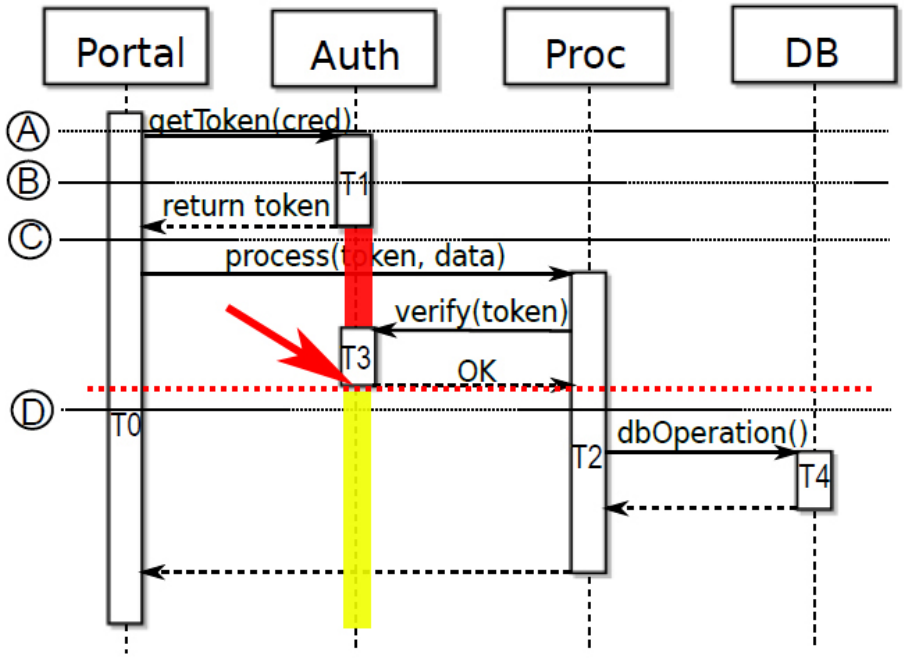


Figura 3.4: Istante temporale in cui Auth è aggiornabile utilizzando Version Consistency.

la Version Consistency non permette l'aggiornamento di Auth tra T_1 e T_3, garantendo quindi la consistenza del sistema dopo l'aggiornamento, a differenza della Tranquillity. La Version Consistency ci permette anche di guadagnare tempo di aggiornamento e disruption rispetto alla Quiescence, poichè non è più necessario aspettare la passivazione di Portal (al termine della transazione T_0) e di Proc (terminazione di T_2).

3.2 Version Consistency: definizioni

La policy di version consistency, al contrario di Quiescence e Tranquillity, è sufficiente per assicurare riconfigurazioni sicure (che garantiscano correttezza del sistema), con bassa disruption e localmente verificabili, senza cioè la necessità di avere un sistema centralizzato per la gestione dell'algoritmo di riconfigurazione. Prima di presentare l'algoritmo vero e proprio sono necessarie alcune definizioni.

3.2.1 Version Consistency

Una transazione T è version consistent rispetto ad un update $< \sum, \omega, \omega', \Gamma, s >$ se e solo se non esistono due transazioni distine T_1,T_2 che fanno parte dello stesso extended transaction set ext(T) *tali che il componente che esegue T_1 utilizza la vecchia versione ω, mentre il componente che esegue T_2 utilizza la nuova versione ω'. Una riconfigurazione dinamica causata da un aggiornamento $< \sum, \omega, \omega', \Gamma, s >$ è version consistent se tutte le transazioni eseguite dalla configurazione corrente $\sum$ sono version consistent.*

La definizione è giustificata dall'assunzione che la nuova configurazione sia corretta e dal fatto che ogni transazione ancora in esecuzione, insieme a tutte le sue sottotransazioni (dirette e indirette), è eseguita interamente o sulla vecchia o sulla nuova configurazione; inoltre, una transazione che finisce prima (o comincia dopo) l'update non può avere sottotransazioni dirette o indirette eseguite dalla nuova (vecchia) versione di un componente che si sta aggiornando.

Ritornando al nostro esempio, se l'update di Auth avviene dopo che è iniziata la transazione T_0 ma prima che essa mandi la richiesta getToken ad Auth (istante di tempo (A)), allora tutte le transazioni appartenenti ad *ext*(T_0) (cioè tutte le transazioni della figura 2.2) sono servite nello stesso modo in cui sarebbero servite se l'aggiornamento avvenisse PRIMA che tutte cominciassero.
Se invece l'update avviene dopo che Auth ha risposto alla richiesta di verifica da parte di Proc (tempo (D)), tutte le transazioni in *ext*(T_0) sono servite nello stesso modo in cui sarebbero servite se l'aggiornamento avvenisse DOPO che tutte finissero.
Tuttavia, se l'aggiornamento avviene al tempo (C), allora la transazione T_1 è

servita da Auth mentre T_3 da Auth': siccome T_1 e T_3 appartengono entrambi all'extended transaction set $ext(T_0)$, T_0 in questo caso non sarebbe version consistent!

3.2.2 Dipendenze dinamiche: archi future e past

Siccome la version consistency non è direttamente verificabile, per fare questo si inseriscono le informazioni relative alle dipendenze dinamiche aggiungendo archi decorati alla configurazione statica di figura 2.1: questi archi sono aggiunti e rimossi dinamicamente durante l'esecuzione e sono etichettati con future o past.

- un arco **future** rappresenta la possibilità per il nodo sorgente di iniziare una transazione sul nodo target;
- un arco **past** indica che una transazione iniziata dal nodo sorgente è già stata eseguita dal nodo target.

La notazione usata è $C \xrightarrow[T]{future} C'$ ($C \xrightarrow[T]{past} C'$) per indicare un arco future (past) che va dal nodo C al nodo C', etichettato con la transazione root T.

3.2.3 Configurazione valida

Una configurazione statica decorata con gli archi rappresentanti le dipendenze dinamiche tra componenti è valida se gli archi future e past sono creati e rimossi a runtime, secondo questi vincoli:

- (HOST-VALIDITY) *Il componente C che esegue una transazione T deve presentare una coppia di archi $C \xrightarrow[T]{future} C$ e $C \xrightarrow[T]{past} C$ (autoanelli) durante tutta l'esecuzione della transazione T;*
- (LOCALITY) *Ogni arco future $C \xrightarrow[T]{future} C'$ (o past $C \xrightarrow[T]{past} C'$) può esistere solo se esiste un corrispondente arco statico $C \xrightarrow{static} C'$ che collega gli stessi due nodi;*
- (FUTURE-VALIDITY) *Un arco future $C \xrightarrow[T]{future} C'$ deve essere aggiunto prima che la prima sotto-transazione $T' \in ext(T)$, $T' \neq T''$ sia iniziata e non può essere rimosso prima che la transazione eseguita da C inizierà una nuova transazione $T'' \in ext(T)$ su C';*
- (PAST-VALIDITY) *Un arco past $C \xrightarrow[T]{past} C'$ deve essere aggiunto alla fine di una transazione T' eseguita da un nodo C dipendente staticamente e non può essere rimosso almeno finchè T è terminata.*

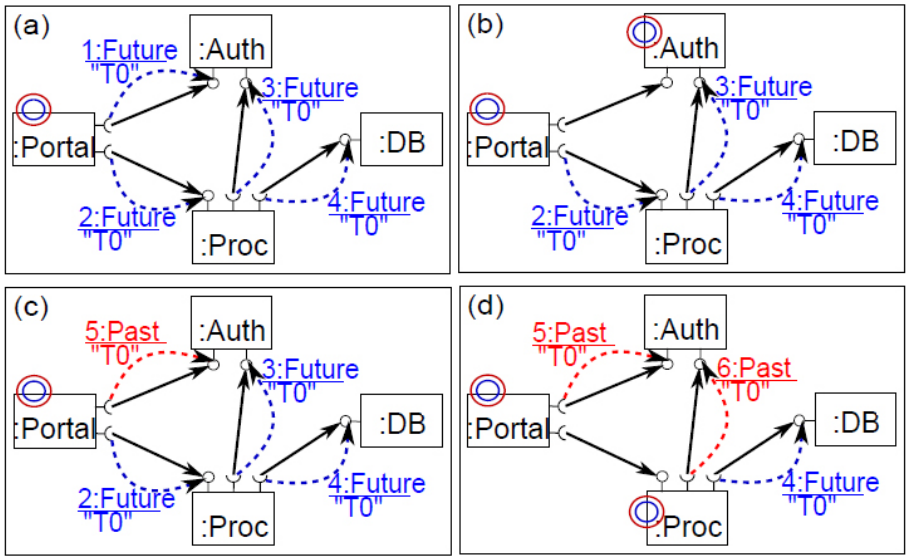


Figura 3.5: Configurazione dinamica negli istanti temporali (A) (B) (C) (D) del sistema di esempio.

La figura 3.5 mostra alcuni esempi di configurazione dinamica valida del nostro sistema (due cerchi concentrici indicano gli autoanelli di un nodo che sta eseguendo una transazione identificata dalla transazione root T_0); le configurazioni in figura fanno riferimento ad alcuni istanti temporali di figura 2.2.
Al tempo (A) la transazione T_0 è appena cominciata su Portal; gli archi dinamici indicano che (i) potrebbe essere eseguita una transazione appartenente a $ext(T_0)$ su Portal; (ii) Portal potrebbe usare Auth e Proc nel futuro, e Proc potrà usare a sua volta Auth e DB.
Al tempo (B) si evidenzia che la transazione T_1 è in esecuzione su Auth ma Portal non eseguirà mai più altre transazioni di $ext(T_0)$ che possano iniziare sottotransazioni su Auth: questo si vede dal fatto che tra i due nodi in questione non ci sono transazioni etichettate con la transazione root T_0.
Al tempo (C) Auth ha (potrebbe aver) terminato delle transazioni in $ext(T_0)$ iniziate in passato da Portal.
Al tempo (D) si nota che Auth non sta eseguendo e non eseguirà mai più transazioni di $ext(T_0)$.

3.2.4 Freeness

Data una configurazione valida, dobbiamo però avere una condizione, verificabile localmente sui nodi, che ci indica se il nodo è version consistent, ovvero se è possibile aggiornarlo in modo sicuro.

Data una configurazione $\sum$, un componente C (o un insieme di componenti ω) è detto libero (free) da dipendenze rispetto alla transazione T, se e solo se non esiste una coppia di nodi future/past etichettati con T in ingresso a C (o ω). C (o ω) è detto free in $\sum$ se e solo se è free rispetto a tutte le transazioni nella configurazione.

Auth è libero da dipendenze rispetto a T_0 nella configurazione di figura 3.5 negli istanti (A) e (D), mentre non lo è in (B) e (C). Intuitivamente, per una configurazione valida $\sum$, la freeness di un componente C, rispetto alla transazione root T, significa che le relative transazioni distribuite modellate da *ext*(T) non hanno ancora usato C oppure non useranno mai più C. Questa considerazione ci porta alla seguente proposizione: *Data una configurazione valida $\sum$ di un sistema, una riconfigurazione dinamica di un insieme dei suoi componenti ω è version consistent se è effettuata quando ω è free in $\sum$.*

3.3 Algoritmo di gestione delle dipendenze dinamiche

Un vantaggio di specificare le dipendenze dinamiche con archi future e past è che la validità di una configurazione può essere raggiunta tramite la cooperazione dei componenti. Ogni componente può prendere la sua decisione localmente senza dover per forza avere informazioni sulla logica dell'applicazione che ospita; chiaramente anche la verifica della freeness viene effettuata in loco dal componente che deve essere aggiornato.
Grazie alla definizione di *configurazione valida*, un nodo può ridurre l'overhead mantenendo in memoria solamente gli archi dinamici che permettono al sistema di essere valido a runtime: si potrebbe chiaramente far finta di nulla mantenendo tutti gli archi relativi ad un transazione root T, creando tutti gli archi all'inizio della transazione T e rimovendoli solo alla fine di essa; tuttavia, anche se la version consistency sarebbe assicurata, il tempo di disruption potrebbe risultare troppo alto.
Assumiamo che data una transazione T, il componente host che la esegue h_T conosca (in un determinato istante di tempo):

- $f(T)$: l'insieme degli archi statici attraverso i quali potrà (nel futuro) iniziare sotto-transazioni sui componenti host vicini;
- $p(T)$: l'insieme degli archi statici attraverso i quali ha (nel passato) iniziato sotto-transazioni sui componenti host vicini.

La configurazione globale del sistema con le dipendenze dinamiche è mantenuta in modo distribuito.
Ogni componente ha solo una conoscenza locale della configurazione che include sè stesso e i nodi vicini; ogni componente è responsabile della creazione/rimozione

di archi dinamici in uscita, ma è anche notificato della creazione, da parte dei vicini, degli archi dinamici in ingresso a se stesso.

3.3.1 Algoritmo

L'algoritmo di gestione delle dipendenze dinamiche è applicato indipendentemente per transazioni diverse, dato che gli archi sono identificati dalla corrispondente root transaction. Se quindi consideriamo un insieme di transazioni distribuite *ext(T)*, l'algoritmo consiste di tre passi: **set up**, **progress**, **clean up**.
Durante l'esecuzione dell'algoritmo, la local-validity della configurazione è assicurata creando solamente archi dinamici che si accoppiano con quelli statici già esistenti, e la host-validity è preservata creando sempre autoanelli locali future/-past appena è iniziata una transazione e cancellandoli appena essa è terminata (se non ci sono altre transazioni correnti che necessitano di questi archi). Nel seguito descriveremo in dettaglio i tre passi dell'algoritmo, mentre nell'appendice A è possibile consultare il relativo pseudo-codice.

3.3.1.1 Passo 1: Set up

Questo passo è eseguito appena è iniziata la root transaction T e prima che essa dia il via a sub-transactions; durante questa fase l'host component h_T crea un arco future per ognuno dei suoi archi statici in uscita che T potrebbe usare per iniziare sotto-transazioni, andando a leggere il contenuto dell'insieme $f(T)$; dopo aver creato un arco future, h_T notifica il vicino interessato e aspetta un ack di risposta: solo dopo h_T avrà il via per poter iniziare una possibile sotto-transazione.
Appena un componente C riceve la notifica della creazione di un arco in ingresso $fe = C' \xrightarrow[T]{future} C$, esso stesso comincia a creare i propri archi future, notificando i suoi vicini e aspettando gli ack di risposta, e infine notificando C' con il proprio ack. Il ragionamento usato è il seguente: il componente C, accettando la creazione dell'arco fe, promette a C' che eseguirà qualche transazione $T_C \in ext(T)$, ma per confermare la propria promessa deve prima aspettare le conferme dai componenti figli che T_C potrà usare a sua volta.
In questo modo, ricorsivamente tutti i componenti interessati da una data transazione root T creeranno i propri archi future. La figura 3.6 mostra il risultato della fase di setup nel nostro sistema di esempio.

3.3.1.2 Passo 2: Progress

Durante questa fase si gestisce la configurazione dinamica durante l'esecuzione delle transazioni in $ext(T)$: gli archi future sono rimossi gradualmente appena l'algoritmo è a conoscenza che il componente *non-userà-più* un altro, e gli archi past sono creati per registrare che il componente *ha-usato* un altro.

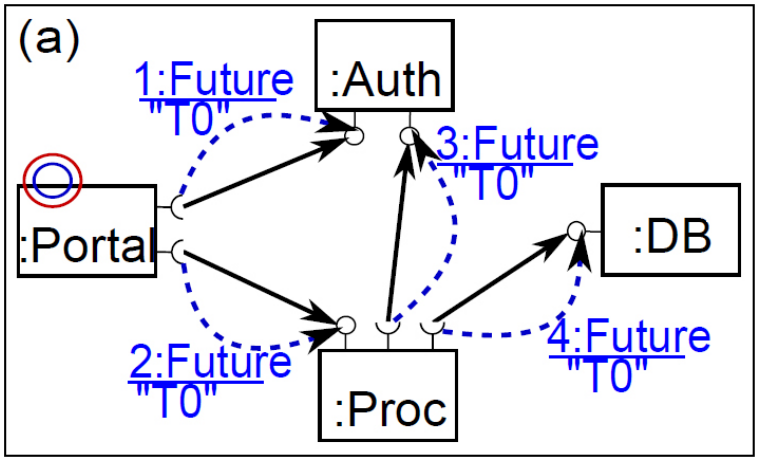


Figura 3.6: Configurazione dinamica dopo la fase di set up.

Più in dettaglio, l'informazione *non-userà-più* può essere disponibile in vari istanti temporali:

1. quando una transazione in $ext(T)$ eseguita dal componente C inizia una sottotransazione in un componente vicino;

2. quando una transazione in $ext(T)$ eseguita dal componente C termina;

3. quando C è notificato della rimozione di un arco future in ingresso, etichettato con T.

Una volta ricevuta l'informazione (non ci interessa quando), un arco future (che non sia autoanello) $fe = C \xrightarrow[T]{future} C'$ è rimosso solo quando:

1. non c'è nessuna transazione $T' \in ext(T)$ in esecuzione sul componente C che inizierà una qualsiasi sottotransazione su C' attraverso l'arco statico $C \xrightarrow{static} C'$;

2. non c'è nessun arco future etichettato con T in ingresso a C, cioè nessuna transazione $T'' \in ext(T)$ sarà più eseguita su C.

Per registrare l'informazione *ha-usato*, quando termina una sotto-transazione T iniziata da T', viene creato immediatamente un arco past $pe = h'_T \xrightarrow[root(T)]{past} h_T$. Per assicurare past-validity, la creazione dell'arco past è fatta in modo che h_T prima notifichi h'_T della terminazione di T, e poi rimuovendo il corrispondente arco locale solo quando pe è stato creato da h'_T.

Nel nostro scenario, Portal rimuove l'arco future su Auth dopo che aver iniziato T_1 su di esso, essendo sicuro che T_0 non inizierà più transazioni simili (figura 3.7);

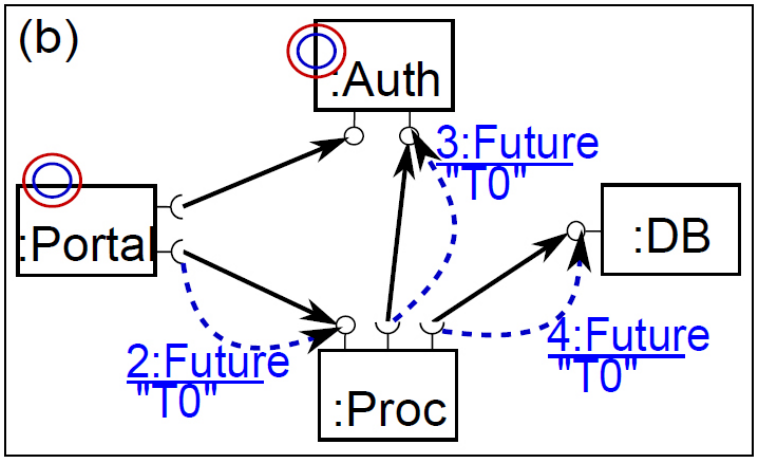


Figura 3.7: Configurazione dinamica dopo la rimozione del primo arco future.

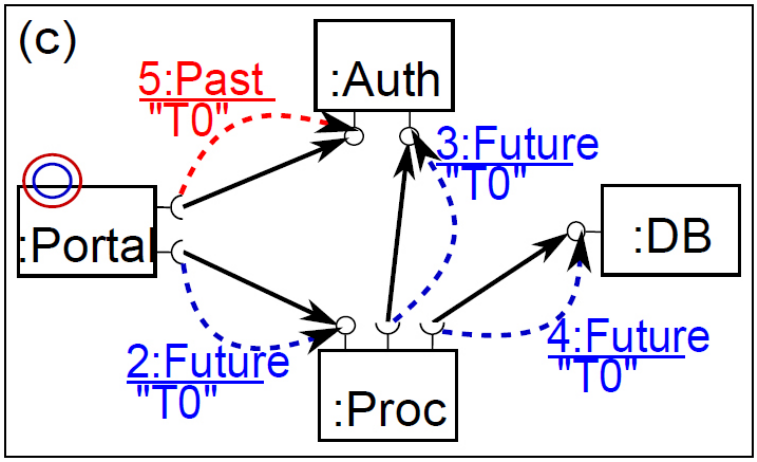


Figura 3.8: Configurazione dinamica dopo la creazione del primo arco past.

quando T_1 finisce, portal crea immediatamente un arco past per registrare il fatto che ha usato Auth (figura 3.8);
dopo un po il sistema raggiunge la configurazione di figura (figura 3.9), dove Auth raggiunge la freeness rispetto alla transazione root T_0.

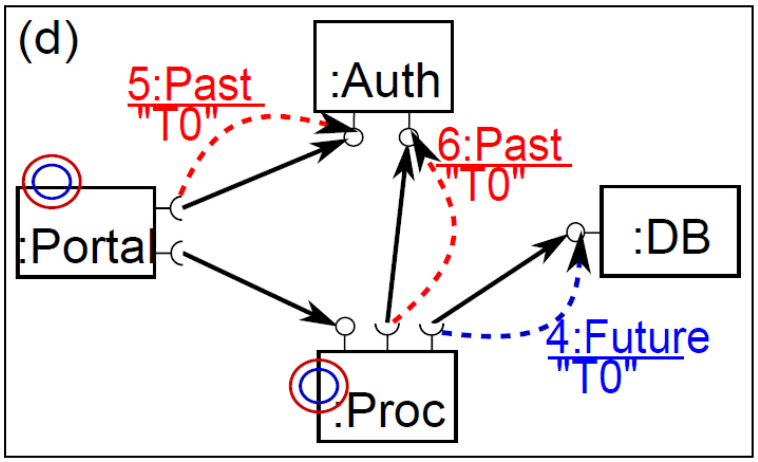


Figura 3.9: Configurazione dinamica quando Auth raggiunge la freeness.

3.3.1.3 Passo 3: Clean up

L'ultimo passo dell'algoritmo è effettutato solo quando T termina : si rimuovono ricorsivamente tutti gli archi future e past rimanenti; questo step non affligge la validità della configurazione.

3.3.2 Raggiungimento della freeness

Data una configurazione valida del sistema, la condizione di freeness è verificabile localmente su ogni nodo, ma esistono più strategie per il suo raggiungimento: le discutiamo di seguito.

3.3.2.1 Waiting for freeness

La prima strategia, chiamata *waiting for freeness (WF)*, è molto semplice ed opportunistica: il sistema aspetta che la freeness si manifesti spontaneamente sui nodi (o su un set di nodi ω). Questa strategia ha il vantaggio di non prevedere overhead aggiuntivo nella computazione dell'algoritmo di gestione degli archi dinamici; tuttavia ha uno svantaggio: anche se tutte le transazioni sul sistema ad un certo punto terminano, può capitare che la freeness di ω non si verifichi mai - per esempio, possono sempre esserci transazioni in esecuzione su un componente di ω, oppure si verifichi ma solo dopo un lungo lasso di tempo.

3.3.2.2 Concurrent Versions

Questa strategia, chiamata *concurrent versions (CV)*, permette aggiornamenti più rapidi: il sistema permette ai componenti di ω (vecchia versione) e ω' (nuova versione) di co-esistere durante il processo di update.
In questo modo il sistema può scegliere quale versione del componente può servire una richiesta e decidere quando il componente può essere rimosso: data una configurazione valida, si può scegliere un componente $C \in \omega$ per servire le richieste che arrivano da una transazione T se e solo se C ha già un arco past in ingresso etichettato con la $root(T)$ (quindi continuare ad usare la vecchia versione per le transazioni ancora in esecuzione) e usare i componenti in ω' per servire tutte le nuove richieste.

3.3.2.3 Blocking for freeness

L'ultima strategia viene incontro al fatto che alcuni sistemi non supportano o preferiscono non utilizzare la co-esistenza di versioni multiple: questa strategia si chiama *blocking for freeness (BF)*. Si fa in modo che alcune richieste in ω siano temporaneamente bloccate per evitare la creazione di nuovi archi past etichettati con nuove root transactions; questi componenti diventeranno free quando tutte le transazioni in esecuzione termineranno, e a quel punto sarà possibile aggiornare i componenti e permettere l'esecuzione di tutte le nuove transazioni in coda.

In conlcusione, la strategia CV è quella preferibile, poichè non introduce overhead (a differenza della BF) e permette di raggiungere la freeness in tempi brevi (a differenza della WF), ovvero permette di avere bassa disruption. Per quanto riguarda la timeliness, CV e BF sono sostanzialmente equivalenti.

Capitolo 4

Version Consistency nei processi di business

4.1 Il contesto tecnologico

La nostra tesi consiste essenzialmente nel calare il criterio di Version Consistency in un CDBS composto da processi di business. Nei prossimi paragrafi faremo una panoramica delle tecnologie esistenti in questo preciso contesto, che abbiamo poi sfruttato per effettuare la nostra implementazione (descritta subito dopo).

4.1.1 Software come servizio (SaaS)

Negli ultimi anni si è verificata una crescita vertiginosa ed una diffusione globale del web che ha modificato radicalmente anche le metodogie di progettazione e di fruizione del software. Le aziende e le organizzazioni stanno forzatamente diventando sempre più e-company e non possono fare a meno di inserirsi e di sfruttare questo "mondo" in continua crescita. Internet è uno strumento fondamentale che fornisce un concreto supporto per il perseguimento degli obiettivi di business. In questo contesto, i sistemi IT aziendali devono essere predisposti per sfruttare al meglio le potenzialità e le nuove tecnologie, con particolare attenzione all'ottimizzazione dei costi e ai ritorni degli investimenti.

Il **SaaS**, acronimo di *"Software as a Service"*, è un modello di sviluppo e di distribuzione del software in continua crescita in cui il concetto di *servizio* assume un ruolo fondamentale.
Nel modello Saas il software viene distribuito sul web come servizio e l'utente (che può essere indistintamente un privato, un'azienda o chiunque sia interessato a seconda del target previsto dal servizio) può accedere remotamente alle diverse funzionalità previste.

Semplificando il concetto, il metodo SaaS prevede che un produttore di software sviluppi, operi (direttamente o tramite terze parti) e gestisca un'applicazione

web che mette a disposizione via Internet ai clienti interessati. L'aspetto interessante consiste nel fatto che i clienti non pagano più per il possesso della licenza del software,ma soltanto per l'utilizzo dello stesso quando necessario.
Il modello SaaS garantisce quindi notevoli vantaggi alle aziende, che possono utilizzare funzionalità di business a costi decisamente inferiori. Inoltre il software viene gestito remotamente dal fornitore, quindi l'impresa non deve effettuare alcun investimento hardware e viene scaricata dagli oneri infrastrutturali e di gestione, che saranno a carico dei gestori del servizio.
Le soluzioni offerte dai servizi web saranno sempre aggiornate ed eventuali nuove implementazioni saranno più rapide e trasparenti all'utilizzatore; il web inoltre permette agli sviluppatori di avere un bacino di utenza universale per i propri servizi, fornendo funzionalità flessibili e personalizzate.
Il provider solitamente dispone di un'architettura centralizzata che consente di fornire le funzionalità software a tutti i vari utenti con una piattaforma comune. Ovviamente poi si potranno sviluppare e prevedere diverse configurazioni per soddisfare le esigenze e le preferenze differenti dei vari clienti: anche il fornitore quindi può godere di una riduzione dei costi dell'infrastruttura, in quanto questa viene spesso condivisa, ed inoltre può dedicarsi al conseguimento di economie di scala.

4.1.2 L'architettura SOA

La diffusione di software fruibile come servizio web ha avuto come conseguenza logica lo sviluppo di nuove architetture e metodologie per la progettazione dei sistemi informatici. In particolare è stato definito un nuovo modello logico che sta acquisendo importanza nello sviluppo del software, definito come **SOA** (*Service Oriented Architecture*): l'architettura dei sistemi è sempre più orientata ai servizi Web, che offrono scalabilità e flessibilità.

É interessante analizzare la definizione formale fornita dall'OASIS (*Organizzazione per lo sviluppo di standard sull'informazione strutturata*):

L'architettura SOA è un paradigma per l'organizzazione e l'utilizzazione delle risorse distribuite che possono essere sotto il controllo di domini di proprietà differenti. Fornisce un mezzo uniforme per offrire, scoprire, interagire ed usare le capacità di produrre gli effetti voluti consistentemente con presupposti e aspettative misurabili.

Questa architettura consente dunque di utilizzare singole applicazioni (o singoli servizi) come componenti di un processo di business, componendo il sistema in modo da soddisfare in modo dinamico le proprie esigenze o le esigenze degli utenti.

La rivoluzione legata all'introduzione di SOA è rappresentata dal fatto che il processo di business non è più vincolato ad una specifica piattaforma o ad un'ap-

plicazione, ma può invece essere considerato come un componente di un processo più ampio che può essere modificato dinamicamente. La globalizzazione e l'evoluzione rapida e continua delle tecnologie dell'informazione obbligano le imprese ad innovare costantemente i propri processi di business, quindi la flessibilità offerta da un'architettura a servizi assume un ruolo determinante: un insieme di servizi "composti" e fatti cooperare tra loro in modo da creare una soluzione funzionale per realizzare gli obiettivi del processo di business aziendale.

I *Web Services* (di cui parleremo più diffusamente in seguito) sono una componente fondamentale di un'architettura SOA, poiché si prestano in modo ottimale per essere utilizzati come semplici servizi. Essi offrono un'interfaccia software che può essere utilizzata dagli utenti per la fruizione di tali servizi; a seguito di una richiesta il web service fornisce il risultato previsto in base al tipo di operazione che deve essere svolta mentre la logica di business legata a tale operazione è totalmente trasparente all'utente.

I servizi web devono possedere alcune caratteristiche fondamentali per essere adatti ad integrarsi all'interno di un'architettura a servizi:

- devono essere ben definiti ed indipendenti da altri servizi, in modo che risulti sempre agevole la fruizione e l'integrazione anche all'interno di contesti complessi;
- devono presentare un'interfaccia precisa che sia indipendente dall'implementazione, che deve risultare totalmente trasparente. I metodi e le tecnologie utilizzate per l'implementazione devono essere gestiti dal fornitore, all'utente interessa soltanto il lato funzionale. Hanno rilevanza le funzionalità disponibili, non la conoscenza dei dettagli tecnici dell'implementazione;
- devono essere ricercabili ed integrabili in modo da poter collaborare e cooperare con altri servizi all'interno di un processo.

La realizzazione di un'architettura SOA, essendo basata sulla composizione di sistemi tramite l'integrazione di servizi, prevede l'interazione tra diversi attori. Oltre ai fornitori e ai fruitori dei vari servizi è importante anche il ruolo di chi permette di trovare i servizi stessi; gli attori principali coinvolti sono i seguenti:

- Service Provider: realizza e mette a disposizione il servizio, rispettando le caratteristiche dei Web Service evidenziate in precedenza. Possono decidere di pubblicare il proprio servizio all'interno di un apposito registry in modo da favorirne la ricerca.
- Service Broker: si occupa della gestione del registry dei servizi, che permette la ricerca di un servizio sulla base delle caratteristiche (funzionalità, performance) con le quali è stato definito e memorizzato nel registry. É possibile anche mettere in atto politiche sulle interrogazioni degli utenti in modo da limitare l'accessibilità ai servizi o personalizzarne le ricerche.

- Service Requestor: è l'utente che richiede il servizio; può interagire con il Service Broker per ottenere il servizio più adatto ai propri obiettivi. Una volta individuati il servizio e il fornitore adatto, il richiedente si collega al Service Provider corrispondente e può iniziare ad usufruire delle operazioni di interesse.

Questi attori possono essere distribuiti fisicamente in ambito globale e possono utilizzare piattaforme tecnologiche completamente diverse; l'unica condizione è che utilizzino un canale trasmissivo comune (che nel nostro caso è rappresentato dal web).

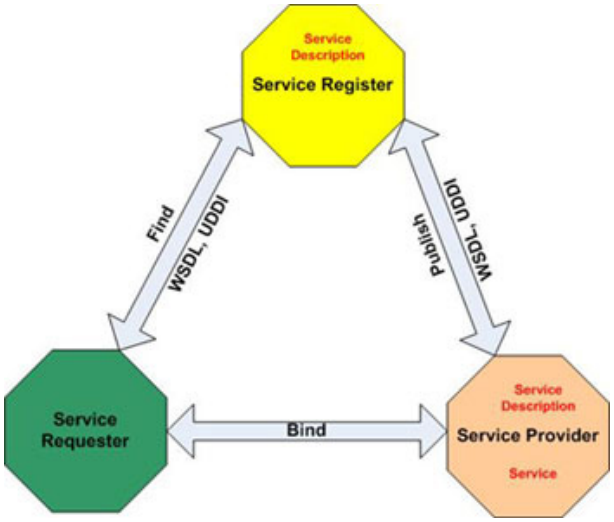


Figura 4.1: Service Oriented Architecture

Le imprese o più in generale gli sviluppatori, una volta definita l'analisi dei requisiti, sfruttando l'architettura SOA possono effettuare composizioni di servizi per realizzare la parte fisica e tecnologica dei sistemi e per realizzare i propri processi di business. In questo modo possono usufruire di tutti i vantaggi legati a questa particolare metodologia di progettazione: riutilizzo di funzionalità esistenti, minimizzazione dei costi legati alle infrastrutture e all'hardware, alta flessibilità e possibilità di integrazioni, massima scalabilità e dinamicità.

In un'architettura SOA ideale i servizi dovrebbero avere un alto livello di *"disaccoppiamento"*, in modo da non dipendere l'uno dall'altro e da risultare totalmente intercambiabili per garantire flessibilità e scalabilità al sistema; al contrario è positivo che ci sia un basso livello di *"coesione"* in modo che ogni servizio svolga una funzionalità ben precisa (che sarà parte del processo di business).

Per garantire lo svolgimento della funzionalità il web service può esporre un'interfaccia di tipo ***coarse-grained***, che prevederà un'unica operazione in cui si svolgerà tutta la la logica di business, oppure un'interfaccia di tipo ***fine-grained***, che

prevederà diverse operazioni separate per lo svolgimento della funzionalità; la logica di business verrà quindi suddivisa tra le varie operazioni disponibili.
In generale è preferibile che un servizio possieda un'interfaccia di tipo *coarse-grained* poiché la frequenza dello scambio di messaggi tra richiedente e web service è un parametro fondamentale ai fini delle performance. Un'unica operazione esposta dal web service equivale ad una singola richiesta; ciò implica una maggiore efficienza poiché il numero di messaggi scambiati è presumibilmente minore rispetto ad un servizio strutturato con interfaccia di tipo *fine-grained*. La "granularità" del servizio può quindi influire sulle prestazioni.

Questi concetti (*coesione*, *disaccoppiamento*, *granularità*) relativi alle caratteristiche dei servizi che compongono un sistema vengono spesso utilizzati anche come metriche per misurare il livello di qualità e di performance di architetture service-oriented. Esistono formule matematiche precise che prendono in considerazione questi parametri permettendo ai progettisti di effettuare delle analisi approfondite. Ciò mette in evidenza che il concetto di architettura SOA è sì molto importante, ma la qualità e le prestazioni dei sistemi dipendono soprattutto dall'efficacia della *"composizione"* e dalla qualità dei servizi stessi che vengono utilizzati.

Quest'ultima considerazione ci permette di introdurre un'altra questione importante: il fornitore di un servizio web ovviamente oltre al corretto svolgimento delle funzionalità previste dovrà garantire anche un determinato livello di performance. Per un cliente oltre ai requisiti funzionali sono determinanti anche i requisiti extra-funzionali ai fini della scelta di un servizio. Alcuni requisiti extra-funzionali fondamentali sono ad esempio il tempo di risposta, laffidabilità o la sicurezza; questi parametri subiscono spesso un'attività continua di monitoraggio e controllo.

I *Service Level Agreement* (SLA) sono strumenti contrattuali attraverso i quali si definiscono i livelli di queste metriche di servizio,che devono essere rispettate dal fornitore nei confronti dei propri clienti. Sono dei veri e propri obblighi contrattuali che in caso di violazioni comportano delle penalità o dei risarcimenti.

4.1.3 I web services

Come accennato nel paragrafo precedente, una tipologia di servizi che si adatta in modo ottimale all'architettura SOA sono i Web Services. Secondo la definizione del W3C (Web Services Architecture Working Group) un Web Service è

Un'applicazione software identificata da un'URI (Uniform Resource Identifier), le cui interfacce pubbliche e collegamenti sono definiti e descritti come documenti XML, in un formato comprensibile alla macchina (specificatamente WSDL). La sua definizione può essere ricercata da altri agenti software situa-

ti su una rete, i quali possono interagire direttamente con il Web Service, con le modalità specificate nella sua definizione, utilizzando messaggi basati su XML (SOAP) scambiati attraverso protocolli Internet (HTTP).

Un Web Service è quindi un'applicazione software in grado di mettersi al servizio di altri sistemi, da cui verrà invocato. Questi sistemi potranno collegarsi e fare richieste al servizio web (sfruttando l'apposita interfaccia), usufruendo in questo modo delle funzioni messe a disposizione.
Con un'architettura SOA che comprende dei Web Services, applicazioni software scritte in diversi linguaggi di programmazione e implementate su diverse piattaforme hardware possono essere tranquillamente integrate e sono in grado di cooperare tra loro. Ciò può avvenire grazie alle interfacce, che vengono esposte pubblicamente. La logica di business che svolge le operazioni previste dal Web Service è totalmente trasparente, così come le tecnologie utilizzate per l'implementazione: si raggiunge in questo modo l'obiettivo dell'*interoperabilità* tra servizi e sistemi differenti.
L'interoperabilità è solamente uno tra i numerosi vantaggi dovuti all'utilizzo dei Web Services. Questo approccio infatti favorisce anche il riutilizzo di infrastrutture ed applicazioni già sviluppate poiché l'integrazione all'interno di un'architettura SOA dovrebbe essere agevole. L'obiettivo da raggiungere è proprio la facilità di utilizzo, in modo da riuscire a realizzare una composizione di servizi per formare sistemi integrati e complessi.

La caratteristica principale che spinge verso l'utilizzo dei Web Services è comunque l'*indipendenza dall'implementazione* del servizio: come già accennato in precedenza, l'interfaccia che un Web Service presenta sulla rete è indipendente dall'implementazione effettiva del servizio; il fornitore potrà apportare modifiche e aggiornamenti alla logica senza che dall'esterno i clienti notino il cambiamento, quello che conta è mantenere inalterata l'interfaccia e le operazioni esposte.
Si ha quindi un alto livello di disaccoppiamento tra l'utente e il Web Service che viene utilizzato: modifiche da una o dall'altra parte possono essere attuate in maniera trasparente mantenendo immutata l'interfaccia esterna. Ciò genera una grande flessibilità che consente di progettare sistemi software complessi costituiti da componenti totalmente svincolati l'uno dall'altro.
I Web Services, dovendo garantire integrabilità e flessibilità, utilizzano prevalentemente una serie di tecnologie standard (già citate nella definizione del W3C presente all'inizio del paragrafo) : XML, WSDL, SOAP, UDDI.

4.1.3.1 Standards

XML *XML* (*eXtensible Markup Language*) è un formato standard per lo scambio dei dati. Non è un vero e proprio linguaggio di programmazione, ma bensì un metalinguaggio attraverso il quale se ne possono creare altri. XML viene utiliz-

zato come linguaggio di annotazione (*Markup*) che permettere di creare gruppi di marcatori, o più precisamente *tag*, personalizzati. Essendo basato sui *tag* è molto simile ad HTML dal punto di vista sintattico ma in realtà è molto differente dal punto di vista pratico.
HTML definisce una grammatica per la descrizione e la formattazione di pagine web o più in generale di ipertesti, mentre XML è un metalinguaggio utilizzato per creare nuovi linguaggi, con lo scopo di descrivere documenti strutturati. HTML ha un insieme predefinito di tag e delle regole ben definite per il loro utilizzo sintattico; XML al contrario consente di definire propri tag personalizzati e di comporli nel modo opportuno.

L'elemento fondamentale di XML sono quindi proprio i tag, ovvero del testo racchiuso tra i simboli < e >, che contiene delle informazioni e che costituisce di conseguenza un metadato. La scelta dei tag può essere effettuata a seconda delle informazioni che interessa rappresentare e che la specifica applicazione dovrà riconoscere. Per maggiore chiarezza,analizziamo un semplice file XML di esempio:

```
<?xml version="1.0"?>
<canzone>
 <titolo>Titolo<\titolo>
 <parole-scritte-da>Autore<\parole-scritte-da>
    <musica-realizzata-da>Musicista</musica-realizzata-da>
    <traccia nome="Prima">
       <tempo>Andante con moto<\tempo>
       <durata>
          <totale ore="0" minuti="3" secondi="22" />
          <note lunghezza="8" valore="e" ottava="2" />
          <note lunghezza="8" valore="g#" ottava="2" />
       <\durata>
    <\traccia>
</canzone>
```

Il documento è costituito da marcatori e dati strutturati secondo un preciso ordine logico, che consiste in una struttura ad albero.
La prima linea del documento identifica lo stesso come un file XML e ne specifica la versione. Successivamente si ha sempre un primo tag, in questo caso <canzone>) ,che assume il ruolo di *"radice"* del documento. I restanti tag specificano il contenuto vero e proprio, con dei tag di apertura e di chiusura che contengono il testo, ovvero il dato. Un elemento XML è tutto ciò che è incluso tra il tag di apertura e il tag di chiusura; tra i due tag si trova il contenuto dell'elemento che può essere costituito:

- da altri elementi (*Element Content*)
- da semplice testo (*Simple Content*)

- da testo inframezzato da altri elementi (*Mixed Content*)
- da nulla (*Empty Content*)

In caso di tag vuoto, la coppia di tag di apertura/chiusura può essere sostituita da un unico tag vuoto con la seguente struttura :

```
<nometag/>
```

La struttura ad albero del documento permette di mettere in relazione tra loro i vari elementi, e queste relazioni determinano il *"modello"* del documento. Gli elementi XML possono contenere anche degli attributi, ovvero delle informazioni aggiuntive che vengono inserite per completare o arricchire le informazioni degli elementi, in maniera simile a ciò che accade nel linguaggio HMTL.

Il motivo che ha portato alla creazione di XML è stata la necessità di avere documenti strutturati e flessibili che potessero essere utilizzati sulla rete web per lo scambio di dati. XML è ormai diventato uno standard, è un formato riconosciuto universalmente per lo scambio di dati. La sua forza è la sua indipendenza da piattaforme o tecnologie, infatti un documento XML può essere letto ed utilizzato da qualsiasi sistema; inoltre è possibile utilizzare questo metalinguaggio per scopi molto differenti, a seconda delle operazioni che verranno eseguite da un' applicazione di fronte agli specifici tag.

Il linguaggio XML è umanamente poco leggibile, infatti è stato pensato soprattutto per essere letto da sistemi artificiali: è quindi un formato ideale per il trasporto di informazioni o per il trasferimento di dati tra applicazioni.
Proprio per questo motivo, essendo adatto a garantire interoperabilità tra applicazioni, il linguaggio XML è utilizzato come standard per lo scambio di dati dei Web Services. Ovviamente una macchina per poter leggere in automatico un documento XML dovrà usufruire di opportune regole di validazione e di uno schema del modello che possa permetterle di interpretare i dati: proprio a questo scopo sono nati DTD (*Document Type Definition*) e XSD (*XML Schema Definition*).

Il **DTD** è un documento attraverso cui si specificano le caratteristiche strutturali di un XML,specificando la composizione del linguaggio del file XML stesso. In particolare si definisce l'insieme degli elementi del documento XML e la struttura ad albero in cui possono essere inseriti. Quindi le relazioni gerarchiche tra gli elementi,l'ordine di apparizione nel documento e l'obbligatorietà o l'opzionalità di tag e attributi sono tutte caratteristiche ben definite. Un DTD permette ad un'applicazione di sapere se il documento XML che sta utilizzando è corretto(o più precisamente valido) o meno.
Sotto è riportato un esempio di Document Type Definition:

```
<!DOCTYPE CATALOG [
```

```
<!ENTITY AUTHOR "John Doe">
<!ENTITY COMPANY "JD Power Tools, Inc.">
<!ENTITY EMAIL "jd@jd-tools.com">

<!ELEMENT CATALOG (PRODUCT+)>

<!ELEMENT PRODUCT
(SPECIFICATIONS+,OPTIONS?,PRICE+,NOTES?)>
<!ATTLIST PRODUCT
NAME CDATA #IMPLIED
CATEGORY (HandTool|Table|Shop-Professional) "HandTool"
PARTNUM CDATA #IMPLIED
PLANT (Pittsburgh|Milwaukee|Chicago) "Chicago"
INVENTORY (InStock|Backordered|Discontinued) "InStock">

<!ELEMENT SPECIFICATIONS (#PCDATA)>
<!ATTLIST SPECIFICATIONS
WEIGHT CDATA #IMPLIED
POWER CDATA #IMPLIED>

<!ELEMENT OPTIONS (#PCDATA)>
<!ATTLIST OPTIONS
FINISH (Metal|Polished|Matte) "Matte"
ADAPTER (Included|Optional|NotApplicable) "Included"
CASE (HardShell|Soft|NotApplicable) "HardShell">

<!ELEMENT PRICE (#PCDATA)>
<!ATTLIST PRICE
MSRP CDATA #IMPLIED
WHOLESALE CDATA #IMPLIED
STREET CDATA #IMPLIED
SHIPPING CDATA #IMPLIED>

<!ELEMENT NOTES (#PCDATA)>

]>
```

Per superare alcuni limiti di DTD, per la definizione della struttura di un documento XML si ricorre sempre più frequentemente ad **XSD**. Quest'ultima è una tecnica più recente ed avanzata che permette di descrivere le regole di validità di un linguaggio XML. Anche in questo caso si ricorre ad una descrizione di elementi e tipi di dato, sfruttando un XML Schema. Rispetto al DTD, questa tecnica rende possibile la definizione di tipi di dato e di vincoli, ammettendo l'ereditarietà e introducendo anche i namespace per risolvere le possibili ambiguità tra elementi di documenti diversi. Eccone un file di esempio:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="urn:books"
            xmlns:bks="urn:books">
```

```
  <xsd:element name="books" type="bks:BooksForm"/>

  <xsd:complexType name="BooksForm">
    <xsd:sequence>
      <xsd:element name="book"
                   type="bks:BookForm"
                   minOccurs="0"
                   maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="BookForm">
    <xsd:sequence>
      <xsd:element name="author"   type="xsd:string"/>
      <xsd:element name="title"    type="xsd:string"/>
      <xsd:element name="genre"    type="xsd:string"/>
      <xsd:element name="price"    type="xsd:float" />
      <xsd:element name="pub_date" type="xsd:date" />
      <xsd:element name="review"   type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="id"   type="xsd:string"/>
  </xsd:complexType>
</xsd:schema>
```

XML è fondamentale poiché su di esso si basa il *WSDL*, il formato utilizzato per la descrizione dei Web Services.
WSDL *WSDL* (*Web Services Description Language*) è un linguaggio basato su XML che viene utilizzato per descrivere in modo completo un Web Service. L'interfaccia di un servizio viene esposta utilizzando proprio un documento WSDL: è quindi evidente che WSDL assume un ruolo fondamentale per l'esistenza stessa dei servizi web e delle architetture SOA. Permette di creare una descrizione basata su XML riguardo il modo in cui interagire con un determinato servizio, stabilendo le operazioni messe a disposizione e le modalità di utilizzo.
In particolare un documento WSDL stabilisce:

- Operazioni offerte dal Web Service
- URL per l'invocazione del Web Service
- I parametri da passare in ingresso
- Formato dei risultati restituiti
- Formato dei messaggi scambiati

Analizzando più nel dettaglio la struttura di un documento WSDL, è evidente che le parti principali sono le seguenti (essendo scritto con formato XML, ovviamente stiamo parlando di tag):

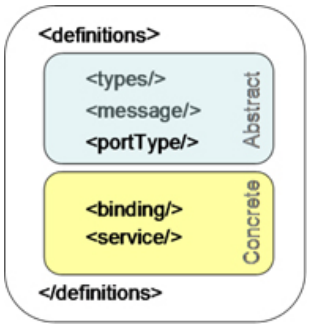


Figura 4.2: Principali tag di un documento wsdl.

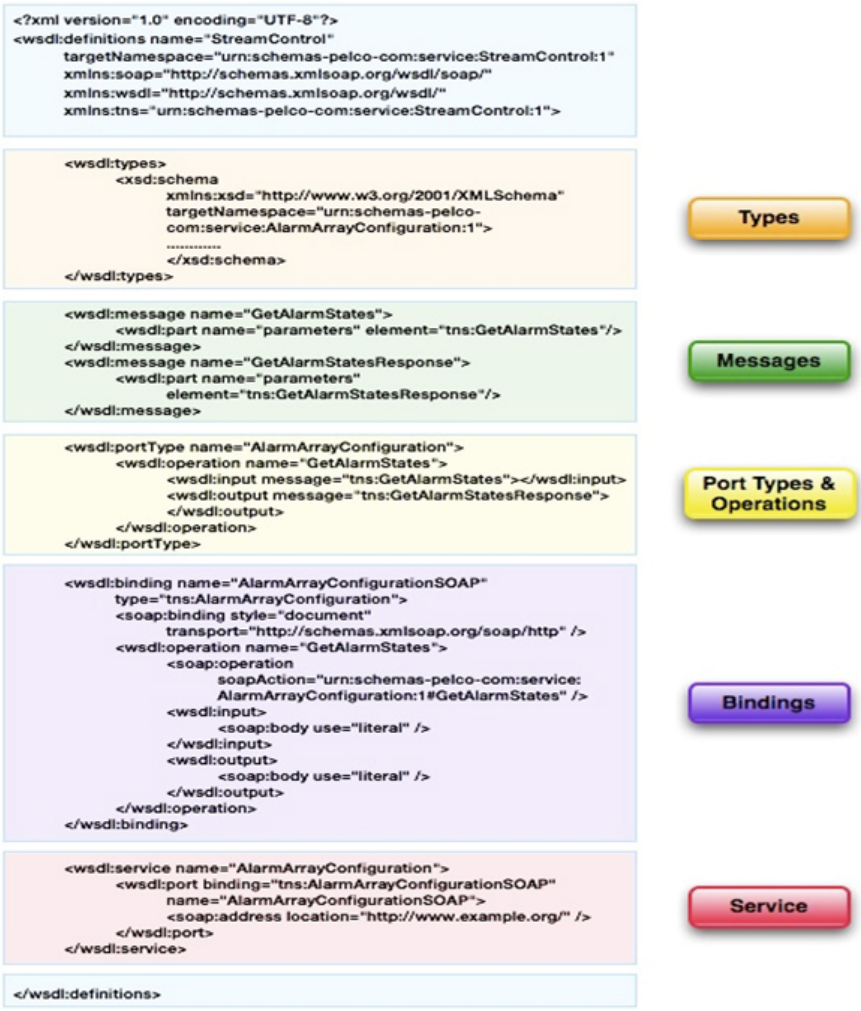
- <types> : contiene la definizione dei tipi di dato utilizzati. Quindi collegandosi a quanto dichiarato in precedenza in questa sezione verrà definito l'XML Schema (o XSD) di riferimento, impostando anche gli appositi namespaces.

- <message> : in questa sezione si definiscono i messaggi utilizzati dal web service per comunicare con le applicazioni client. I messaggi vengono descritti in modo dettagliato, specificando gli elementi da cui sono composti ed eventualmente sfruttando l'XML Schema definito nella sezione precedente. Prendiamo come esempio un semplice Web Service che viene invocato da un client, esegue la sua logica di business e restituisce il risultato previsto dall'operazione richiesta. In questo caso verranno utilizzati due messaggi : con il primo il client invierà la richiesta al WS, mentre il secondo conterrà la risposta e verrà inviato dal WS al client. La struttura di entrambi i messaggi sarà specificata in questa sezione <message> del WSDL.

- <binding> : definisce il formato del messaggio ed i dettagli di protocollo per ogni porta. Viene stabilito il legame di ogni operazione del Web Service al protocollo per lo scambio di messaggi. In particolare si specifica il protocollo per ognuno dei messaggi di un servizio offerto dal Web Service. Il protocollo per lo scambio di messaggi più diffuso è sicuramente SOAP, di cui parleremo nel paragrafo successivo.

- <service> : contiene la localizzazione del Web Service, mostrando lelenco di tutti i servizi messi a disposizione. Per ognuno di essi vengono indicati l'URL e la porta.

SOAP *SOAP* (*Simple Object Access Protocol*) è un protocollo per la trasmissione dei messaggi tra componenti software. É basato anch'esso sul metalinguaggio XML e garantisce un meccanismo semplice (ma nello stesso tempo affidabile) che permette ad un'applicazione di mandare messaggi ad un'altra applicazione. In base alla tipologia di comunicazione prevista si possono avere differenti messaggi,




```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="StreamControl"
        targetNamespace="urn:schemas-pelco-com:service:StreamControl:1"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:tns="urn:schemas-pelco-com:service:StreamControl:1">

        <wsdl:types>
                <xsd:schema
                        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                        targetNamespace="urn:schemas-pelco-
                        com:service:AlarmArrayConfiguration:1">
                        ..........
                        </xsd:schema>
        </wsdl:types>

        <wsdl:message name="GetAlarmStates">
                <wsdl:part name="parameters" element="tns:GetAlarmStates"/>
        </wsdl:message>
        <wsdl:message name="GetAlarmStatesResponse">
                <wsdl:part name="parameters"
                        element="tns:GetAlarmStatesResponse"/>
        </wsdl:message>

        <wsdl:portType name="AlarmArrayConfiguration">
                <wsdl:operation name="GetAlarmStates">
                        <wsdl:input message="tns:GetAlarmStates"></wsdl:input>
                        <wsdl:output message="tns:GetAlarmStatesResponse">
                        </wsdl:output>
                </wsdl:operation>
        </wsdl:portType>

        <wsdl:binding name="AlarmArrayConfigurationSOAP"
                type="tns:AlarmArrayConfiguration">
                <soap:binding style="document"
                        transport="http://schemas.xmlsoap.org/soap/http" />
                <wsdl:operation name="GetAlarmStates">
                        <soap:operation
                                soapAction="urn:schemas-pelco-com:service:
                                AlarmArrayConfiguration:1#GetAlarmStates" />
                        <wsdl:input>
                                <soap:body use="literal" />
                        </wsdl:input>
                        <wsdl:output>
                                <soap:body use="literal" />
                        </wsdl:output>
                </wsdl:operation>
        </wsdl:binding>

        <wsdl:service name="AlarmArrayConfiguration">
                <wsdl:port binding="tns:AlarmArrayConfigurationSOAP"
                        name="AlarmArrayConfigurationSOAP">
                        <soap:address location="http://www.example.org/" />
                </wsdl:port>
        </wsdl:service>

</wsdl:definitions>
```

Figura 4.3: Esempio di documento wsdl.

che hanno in ogni caso una struttura di base comune raffigurata nellimmagine 4.5

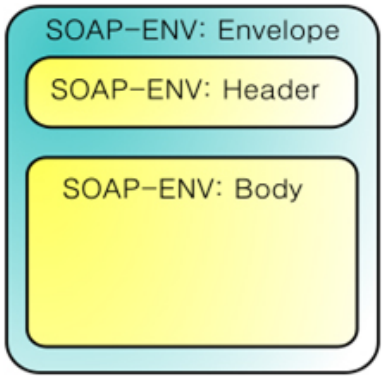


Figura 4.4: Struttura di un messaggio SOAP.

Il messaggio SOAP è composto da un contenitore, chiamato *Envelope*, in cui sono contenute le due sezioni distinte *Header* e *Body*.
L'Header contiene l'intestazione del messaggio, ed è una parte opzionale. In esse possono essere specificate diverse meta-informazioni riguardanti il routing, la sicurezza, le transazioni e altri parametri. Il Body è il corpo vero e proprio del messaggio e in esso si trova l'informazione principale trasportata dal messaggio SOAP dal mittente al destinatario: questo carico informativo viene spesso definito come carico utile o come *payload*.
All'interno di un'architettura SOA, i messaggi scambiati tra i vari Web Services saranno quindi dei messaggi SOAP; nel caso di un messaggio di richiesta il payload del messaggio SOAP conterrà i valori dei parametri necessari per richiedere l'operazione del Web Service (definiti nell'apposito WSDL che espone l'interfaccia esterna), mentre nel caso di un messaggio di ritorno il payload conterrà il risultato dell'operazione restituito dal Web Service.
Un esempio di richiesta SOAP:

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
```

E un esempio di risposta:

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPriceResponse>
    <m:Price>34.5</m:Price>
  </m:GetStockPriceResponse>
</soap:Body>

</soap:Envelope>
```

UDDI I Web Services vengono descritti dal WSDL e comunicano attraverso messaggi SOAP.

UDDI (*Universal, Description, Discovery and Integration*) è un meccanismo che rende possibile la ricerca di Web Services secondo certi criteri come ad esempio la tipologia del servizio. UDDI viene utilizzato sia per pubblicare sia per trovare informazioni sui servizi web, mettendo a disposizione una specie di registro per tali attività; anche questa tecnologia è basata su XML ed utilizza messaggi SOAP per le comunicazioni da e verso l'esterno.
UDDI è stato progettato per essere interrogato automaticamente tramite messaggi SOAP e per fornire direttamente il collegamento ai documenti WSDL che descrivono l'interfaccia dei Web Services. Il registro contiene informazioni sia "human-readable" sia "machine-readable", in modo da poter essere interpretate ed utilizzate automaticamente dai sistemi.

É curioso notare che anche UDDI si comporta esattamente come i Web Services di cui gestisce pubblicazioni e ricerche. Infatti riceve un messaggio SOAP contenente nel body i parametri di richiesta e restituisce in un messaggio SOAP i dati di risposta; inoltre anche in questo caso vengono esposte diverse operazioni che possono essere invocate ed utilizzate.

4.1.4 I processi di business: BPEL

Per realizzare un'architettura orientata ai servizi è necessario essere in grado di coordinare ed integrare i vari Web Service. Proprio a questo scopo è stato creato BPEL, un linguaggio basato sempre sull'XML.

BPEL viene utilizzato per descrivere formalmente dei processi commerciali in modo da permettere e coordinare una suddivisione dei compiti tra attori diversi; esso assume un ruolo di coordinazione di servizi web, permettendo la realizzazione di composizione di servizi proprio come previsto dall'architettura SOA e consentendo agli sviluppatori di definire dei processi di business sfruttando i Web Services: il processo di business sarà composto da una serie di attività, che possono essere semplici o strutturate.
Un'applicazione BPEL viene invocata anch'essa come Web Service ed interagirà

con altri Web Services: per questo motivo anche l'applicazione BPEL possiederà un proprio WSDL ed esporrà pubblicamente tutte le operazioni disponibili per l'utente finale.
Le relazioni con i partner esterni (Web Services con cui interagirà per eseguire la logica di business) sono invece private e totalmente trasparenti all'utente. I partner sono gli attori esterni con cui il processo interagisce e possono sia offrire che richiedere un servizio.

Il tag <partner> nel documento XML del linguaggio BPEL contiene quindi i Web Services con cui il processo interagisce. La relazione tra un partner ed un processo è definita attraverso un <partnerLink>, che a sua volta è un'istanza di un <partnerLinkType> che descrive la relazione tra due servizi generici; questa relazione prevede la definizione dei ruoli: deve essere specificato il ruolo del servizio che fornisce l'operazione.
In caso di comunicazione *sincrona* tra servizi solitamente viene specificato solamente il ruolo di chi viene invocato e svolge l'operazione, mentre nel caso di comunicazione *asincrona* vengono specificati entrambi i ruoli perché anche chi svolge l'operazione dovrà effettuare una chiamata nei confronti del richiedente per restituire il risultato.

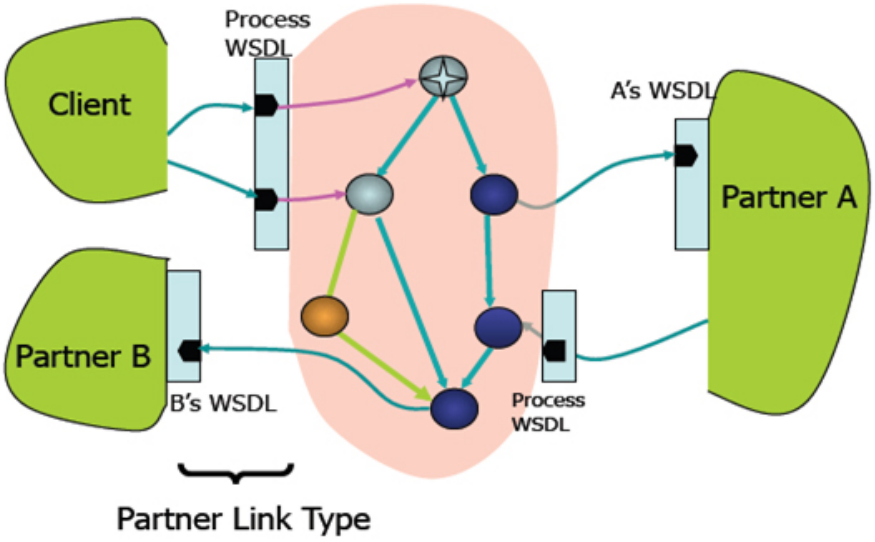


Figura 4.5: Rappresentazione di un processo BPEL.

Ovviamente un processo di business comporterà uno scambio di messaggi e anche la presenza di dati intermedi usati per gestirne la logica; questi elementi definiscono lo stato del processo, che viene mantenuto attraverso le *variabili*. Un processo BPEL quindi contiene anche assegnamenti ed espressioni che agiscono

sulle variabili per modificarne lo stato: le variabili contengono spesso dei dati che vengono poi utilizzati nei messaggi, i quali vengono scambiati con gli altri attori del processo.

La parte fondamentale di un processo BPEL sono sicuramente le attività, che determinano l'esecuzione dell'orchestrazione dei Web Services. Ci sono quattro attività di base che sono quasi sempre necessarie:

- <invoke> : effettua la chiamata ad un partner. Viene specificata l'operazione richiesta, la porta da chiamare, la variabile di input e (nel caso sincrono) la variabile di output;
- <receive> : il processo si mette in attesa per ricevere la chiamata di un partner. Vengono specificati l'operazione che verrà invocata e la porta su cui si verrà chiamati. Dopo una <receive> solitamente viene creata una nuova istanza del processo. In caso di comunicazione sincrona sicuramente una <receive> verrà seguita da una <reply> per restituire l'output al chiamante;
- <reply> : viene utilizzata solamente in caso di comunicazione sincrona per restituire i dati al client;
- <wait> : blocca il processo.

BPEL mette a disposizione dei costruttori per stabilire l'ordine in cui devono essere svolte le varie attività che compongono il processo. I principali sono:

- <sequence> : contiene una o più attività che verranno eseguite in sequenza;
- <switch> : ha lo stesso significato dello "switch" presente nei linguaggi di programmazione. Definisce un gruppo di possibili diramazioni e ciascuna di esse è definita da un <case>. Le condizioni che portano nei vari rami possono non essere mutuamente esclusive perché in ogni caso verrà eseguita solamente il primo ramo con condizione vera;
- <while> : anche in questo caso viene rappresentata la stessa struttura dei linguaggio di programmazione. In caso di condizione vera si ripetono le attività del "while" in un ciclo;
- <flow> : le attività presenti in questo costruttore vengono eseguite in parallelo. Solamente quando tutte le attività terminano il processo BPEL può proseguire;
- <pick> : le attività interne vengono svolte soltanto quando si verificano dei particolari eventi. Il costruttore "pick" quindi mette il processo in attesa di eventi come ad esempio l'arrivo di un messaggio o la scadenza di un timer.

Utilizzando soltanto questi elementi di base presentati, BPEL è già in grado di coordinare in modo interessante le attività di un processo, ad esempio è possibile eseguire semplicemente le attività in sequenza (figura 4.6).


```
<bpws:sequence>
   attività1
   attività2
   attività3
</bpws:sequence>
```

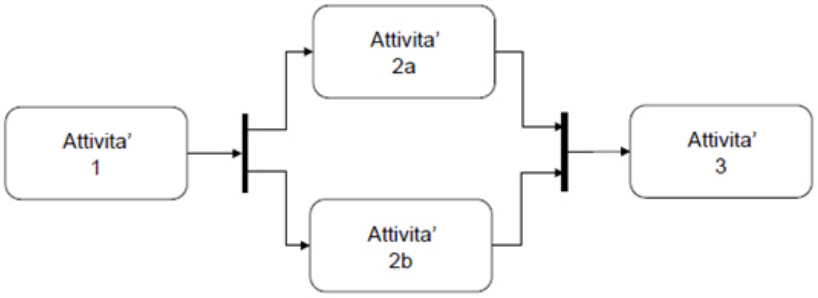
Figura 4.6: Semplice sequenza di attività in un processo BPEL.

É possibile creare una sequenza in cui due attività vengono parallelizzate per poi risincronizzare il processo al termine del parallelismo (figura 4.7), oppure effettuare una scelta in modo da eseguire un'attività o un'altra attività in modo esclusivo, come in figura 4.8.

Un processo BPEL oltre a definire i partner coinvolti nell'esecuzione e le attività previste in essa, presenta anche alcune sezioni per gestire alcune situazione più complesse.
Grazie ai *Correlation Sets* si possono gestire anche interazioni di tipo *"Stateful"*, ovvero transazioni in cui tra una chiamata e l'altra viene mantenuto lo stato del processo attraverso degli appositi dati. In questo modo un messaggio di richiesta sarà in grado di raggiungere l'istanza corretta di un processo già invocato in precedenza, quindi la sezione <correlationSets> contiene un gruppo di dati di proprietà che permettono di effettuare comunicazioni asincrone di tipo stateful.

Un'altra sezione importante è costituita dalle varie tipologie di *handler*. Un processo può infatti disporre di *event handler* per gestire determinati tipi di eventi, di *fault handler* per garantire una corretta gestione degli errori, di *compensation handler* per modificare lo stato di un processo o l'esito di alcune attività svolte in base a determinate condizioni. Per ogni tipologia di handler è prevista una determinata sezione del documento BPEL, quindi potranno essere presenti i tag <eventHandlers>, <faultHandlers>, <compensationHandlers>.

I passi fondamentali per la progettazione di un processo BPEL sono i seguenti:




```
<bpws:sequence>
   attività1
   <bpws:flow>
     attività2a
     attività2b
   </bpws:flow>
   attività3
</bpws:sequence>
```

Figura 4.7: Parallelizzazione di attività in un processo BPEL.


```
<bpws:sequence>
   attività1
   <bpws:switch>
     <bpws:case condition="c1">attività2a</bpws:case>
     <bpws:case condition="c2">attività2b</bpws:case>
   </bpws:switch>
   attività3
</bpws:sequence>
```

Figura 4.8: Branching di attività in un processo BPEL.

1. Identificazione degli attori : bisogna individuare i partner per l'esecuzione e il client che avvierà il processo.

2. Identificazione delle attività e del loro ordine di esecuzione previsto.

3. Ricerca o specifica dei WSDL dei partner, definendo anche se la comunicazione è di tipo sincrono o asincrono.

4. Specifica del proprio WSDL in modo che si possa venire invocati dai client.

5. Definizione dei partner Link e dei partnerLinkType necessari.

6. Definizione della struttura vera e propria del processo BPEL.

4.1.5 BPEL engines

Un processo BPEL, una volta sviluppato, ha bisogno di un apposito engine per essere concretamente eseguito. Sono stati sviluppati diversi motori BPEL che consentono di eseguire i processi, ma in particolare ci occuperemo di **ActiveBpel Engine**, un ambiente runtime completo per processi BPEL, open source e completamente scritto in Java.
L'engine consente di eseguire un processo BPEL e di verificarne la corretta esecuzione. Può quindi essere utilizzato anche per attività di monitoraggio e di controllo dei processi.
In sintesi, la funzione di ActiveBpel è praticamente quella di leggere le definizioni dei processi BPEL e creare da questi le rappresentazioni dei processi stessi. Quando arriva un messaggio il motore crea un'istanza di processo e la esegue prendendosi cura della corretta esecuzione delle attività. L'engine per essere eseguito ha la necessità di essere contenuto in un server (o più precisamente in un servlet container), come ad esempio Tomcat. ActiveBpel è associato anche ad un ambiente di sviluppo proprietario che consente di progettare i processi di business.

Un altro engine molto diffuso è **Apache ODE** (Orchestration Director Engine). É un motore per processi Bpel sviluppato anch'esso in Java, e come ActiveBpel necessita di un server web che supporti le servlet per essere utilizzato.
Apache ODE supporta la comunicazione con i web services, l'invio e la ricezione di messaggi, la gestione dei dati associati ai processi e il recupero degli errori. Supporta definizioni di processi sia nello standard WS-BPEL 2.0 di OASIS sia nello standard BPEL4WS 1.1.
Un aspetto interessante è il supporto da parte di entrambi gli engine di due differenti layer di comunicazione: il più classico basato su Axis2 (che gestisce il trasporto http nei web services) e un altro basato sullo standard JBI.

ActiveBpel e Apache ODE sono ottime soluzioni open source,ma è molto diffuso anche **Oracle Bpel Process Manager**. Quest'ultimo richiede l'acquisto di una licenza e grazie al brand Oracle è molto diffuso a livello aziendale. Anche

questo engine gestisce l'esecuzione di processi di business che coordinano l'orchestrazione di applicazioni e di web services.

Comparando i vari engine, ActiveBpel e Apache ODE (pur essendo open source) spiccano per un maggior supporto nei confronti delle specifiche dello standard OASIS. OracleBPEL presenta un ambiente di sviluppo più completo ma paradossalmente in base a molti benchmark non tiene il passo dei due engine open source anche dal punto di vista delle performance e del tempo di esecuzione dei processi.
Un punto dolente ancora irrisolto consiste nella portabilità dei processi BPEL, che devono essere progettati in base all'engine su cui verranno eseguiti perché spesso non si riesce ad eseguire indistintamente il deploy sui vari engine a causa di differenti adesioni agli standard o di alcune incompatibilità.

Per quanto riguarda la scelta dell'engine adatto alle proprie esigenze, in uno scenario in cui si prospettano alti carichi di lavoro la soluzione migliore potrebbe essere Apache ODE. Se invece si ricercano soprattutto sicurezza ed assistenza nello sviluppo, il marchio Oracle ovviamente è una garanzia sotto questo punto di vista. La soluzione intermedia, ma probabilmente migliore, è ActiveBpel. Quest'ultimo oltre a rispettare ottimamente gli standard BPEL offre prestazioni paragonabili ad Apache ODE ed una buona probabilità di trovare assistenza nelle comunità online di sviluppatori.

4.1.6 I processi BPEL dinamici: Dynamo

Il linguaggio BPEL si è ormai affermato come standard per effettuare composizioni e orchestrazioni di web services o più in generale di componenti software. Gli engine BPEL permettono di gestire in maniera abbastanza basilare alcune caratteristiche avanzate legate all'esecuzione dei processi (come ad esempio la gestione degli errori e la loro compensazione, l'invio di messaggi aggiuntivi, la lettura a runtime di alcuni dati).

Dynamo è una tecnologia sviluppata appositamente per incrementare le capacità di controllo durante l'esecuzione dei processi Bpel sul loro engine.
Offre un costante monitoraggio run-time, che permette di controllare se il sistema si sta comportando nel modo corretto e di gestire gli errori in caso di necessità. C'è anche la possibilità di controllare direttamente l'esecuzione dei processi Bpel, ad esempio sospendendoli, forzandoli alla terminazione, riavviandoli o "intercettando" l'esecuzione durante le varie attività con lo scopo di svolgere del codice aggiuntivo.
Dynamo si appoggia su ActiveBpel integrando e migliorando l'engine con le funzionalità appena esposte, che permettono di eseguire i processi monitorandoli ed integrandoli nel modo opportuno. Queste attività di monitoraggio e di supervisione sono totalmente trasparenti all'engine, essendo esterne ad esso. Inoltre

vengono eseguite direttamente a runtime integrando l'esecuzione di ActiveBpel.

Dynamo consente di sviluppare delle "regole di supervisione", in cui vengono dichiarate le proprietà funzionali ed extra-funzionali che i processi in esecuzione devono rispettare. Queste regole sono composte da:

- Posizione: identifica l'attività Bpel da monitorare;
- Regola di monitoraggio: definisce il vincolo di monitoraggio;
- Parametri di monitoraggio: determina se le regole devono essere eseguite e quando (priorità delle regole);
- Strategia di recupero: definisce le azioni da eseguire per il recupero in caso di anomalie.

Sono stati definiti due linguaggi speciali per stabilire questi aspetti: *WSCol* (*Web Service Costraint Language*) per specificare le regole di monitoraggio e *WSRel* (*Web Service Recovery Language*) per specificare strategie di recupero. Non entreremo nel dettaglio di questi linguaggi perché ci interessano altri aspetti di Dynamo maggiormente legati alla nostra implementazione.

É comunque interessante analizzare l'architettura su cui si basa Dynamo, raffigurata nell'immagine 4.9.

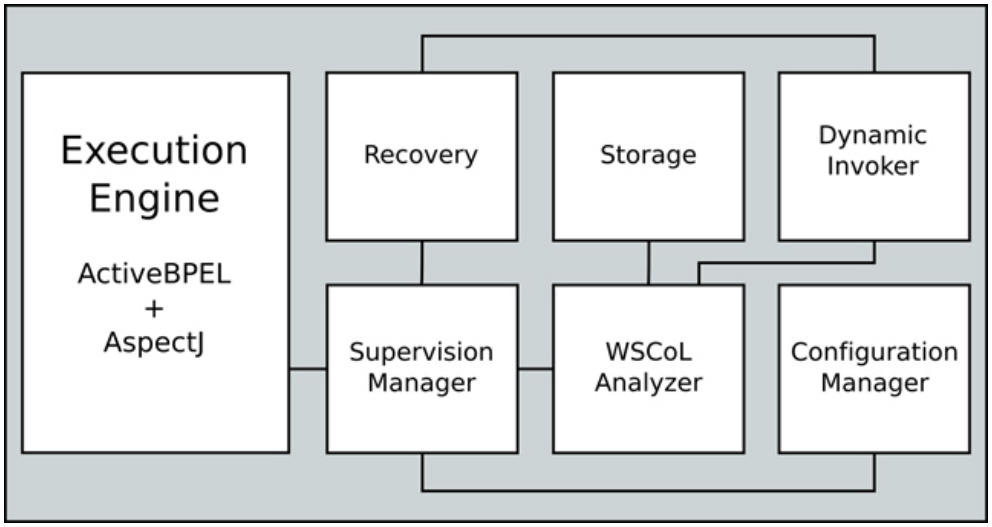


Figura 4.9: Architettura di Dynamo.

I tre componenti principali sono: l'engine di esecuzione dei processi Bpel (ActiveBpel), il sistema di monitoraggio e il sistema di recupero.
Il sistema di monitoraggio consente di interpretare e di verificare regole scritte

nel linguaggio WSCol, definite per una particolare attività del processo.
Il sistema di recupero si occupa invece di eseguire le azioni di compensazione necessarie in caso di errori o di regole violate.
Il Dynamic Invoker è un componente dell'architettura che gestisce le invocazioni SOAP ai web service esterni mentre il Configuration Manager memorizza le regole di supervisione da utilizzare, consentendo il loro inserimento, la loro modifica o la loro cancellazione.

L'engine di esecuzione è leggermente modificato rispetto alla versione originale di Active Bpel poichè sfrutta alcune caratteristiche di AspectJ. In questo modo si possono intercettare le attività che compongono i processi in alcuni istanti precisi, denominati *punti di intercettazione*. É utile soprattutto monitorare l'interazione del processo con il mondo e i componenti esterni quindi i processi vengono intercettati prima e dopo l'esecuzione di attività di Invoke, Receive e Reply.
Durante questi punti di intercettazione, lo sviluppatore, grazie a Dynamo, può inserirsi e gestire il processo monitorandolo, sfruttando le regole di supervisione dichiarate in Dynamo, introducendo codice aggiuntivo, personalizzando il processo, ecc. L'utilizzo di alcune parti di Dynamo sarà un aspetto fondamentale della nostra implementazione che verrà presentata nei prossimi capitoli.

4.2 Lo scenario

In questo progetto di tesi abbiamo implementato il concetto di aggiornamento dinamico in un sistema distribuito sfruttando il criterio di *Version Consistency* e il relativo algoritmo.
Concetti generici presentati in precedenza come ad esempio la configurazione statica, le transazioni, la riconfigurazione dinamica o la freeness sono realmente rimappati nella nostra implementazione concreta. Ovviamente per realizzare questo obiettivo abbiamo dovuto prima sviluppare un modello di sistema distribuito su cui agire con richieste ed esecuzioni di aggiornamenti: era necessario sviluppare un modello di CBDS che si adattasse in modo ottimale all'esecuzione dell'algoritmo, ai fini di evidenziarne la sua validità e la possibilità di utilizzo concreto in ambiti reali.
Per fronteggiare al meglio questa necessità abbiamo scelto di realizzare un CBDS i cui componenti sono dei processi BPEL; tali processi, essendo scritti con il linguaggio BPEL, distinguono perfettamente le varie attività per realizzare le orchestrazioni all'interno di un sistema distribuito.
L'esecuzione a runtime può essere seguita passo dopo passo, anche grazie all'aiuto dell'engine. Nel paragrafo precedente abbiamo presentato Dynamo, che integra l'engine di esecuzione dei processi BPEL (ActiveBPEL) permettendo di intercettare l'esecuzione prima e dopo ogni singola attività ed inserendo dei controlli sui processi.

Questo contesto si è rivelato essere ideale per la nostra implementazione poiché ci ha permesso di seguire passo dopo passo l'esecuzione delle transazioni, inserendo nei punti di intercettazione previsti da Dynamo la nostra logica per la gestione della version consistency. Tale logica è stata sviluppata in una struttura generale esterna molto complessa (che verrà presentata dettagliatamente nei prossimi paragrafi) e permette ai singoli componenti di conoscere in ogni momento se è possibile aggiornare la propria versione o meno.
I punti di intercettazione presenti nell'engine di esecuzione servono soltanto per invocare i metodi della logica (quindi dell'algoritmo di version consistency) che devono essere eseguiti in quel determinato istante di tempo, in base all'attività corrente del processo BPEL e allo stato della transazione.

La logica di business che implementa i concetti generali dell'algoritmo di Version Consistency permette quindi di eseguire aggiornamenti dinamici in modo automatico garantendo la correttezza del sistema e viene interamente gestita attraverso un Web Service esterno. Questa scelta permette di gestire l'esecuzione dell'algoritmo in modo centralizzato, esternamente a Dynamo.
La version consistency si basa sulla conoscenza locale dei singoli componenti del CBDS, che non hanno bisogno di conoscere la configurazione statica e dinamica generale. Proprio per questo motivo abbiamo scelto di associare una singola istanza del Web Service di gestione della Version Consistency ad ogni processo BPEL (che costituisce un componente del CBDS generale). La struttura generale verrà presentata più avanti in modo più chiaro e dettagliato. Nel prossimo paragrafo inizieremo a presentare come abbiamo rimappato i concetti generici di base legati ai CBDS.

4.2.1 Configurazione statica e transazioni BPEL

Il modello di sistema distribuito che abbiamo realizzato è composto da processi BPEL. Questi processi sono gli effettivi componenti del nostro Component Based Distributed System e sarà su di essi che scatteranno le richieste di aggiornamento. Ogni processo BPEL è installato su un server diverso, in particolare su una differente installazione di Tomcat, ciascuna contenente l'engine Active-Bpel. L'utilizzo di più Tomcat, tutti configurati nello stesso modo, ci permette di simulare l'esecuzione delle varie transazioni su un sistema realmente distribuito.

A questo punto siamo in grado di presentare la configurazione statica del nostro sistema, mostrata in figura 4.10, leggermente semplificata rispetto a quella vista nei capitoli 2 e 3.
Il nostro modello è quindi basato su 3 componenti : Portal, Auth e Proc. Sono presenti le seguenti dipendenze statiche:

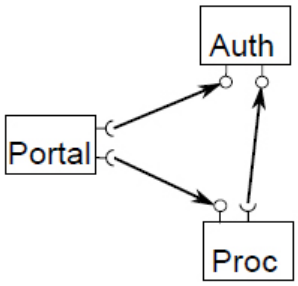


Figura 4.10: Configurazione statica della nostra implementazione.

- dipendenza statica tra Portal ed Auth (Portal può richiedere un servizio ad Auth);
- dipendenza statica tra Portal e Proc (Portal può richiedere un servizio a Proc);
- dipendenza statica tra Proc e Auth (Proc può richiedere un servizio ad Auth).

Ai fini della realizzazione di una transazione, un processo BPEL può effettuare una chiamata ad un altro processo BPEL oppure ad un web service che esegue una determinata logica di business.
Come affermato in precedenza, un processo BPEL rappresenta un componente del sistema. É importante precisare che nel caso in cui un processo BPEL invocasse un web service, il processo in questione e il relativo web service sono considerati come un unico componente ai fini dell'algoritmo di Version Consistency. Quindi sia la modifica della logica di business del web service esterno sia la sostituzione del processo BPEL con una versione più aggiornata costituiscono due esempi di aggiornamento dello stesso componente.

L'obiettivo del nostro sistema ovviamente è l'esecuzione delle transazioni. Per eseguire una transazione completa servirà il contributo di tutti i componenti, quindi al momento dell'esecuzione avverrà un'orchestrazione tramite BPEL. La transazione svolta dal nostro sistema simula un modello di autenticazione distribuito e viene presentata dettagliatamente nel sequence diagram di figura 4.11.
Analizziamo come viene garantito lo svolgimento distribuito dell'intera transazione.
La transazione ha inizio quando Portal riceve una richiesta da un client attraverso una stringa come parametro di ingresso. Al momento dell'invocazione di Portal ha inizio la root transaction T_0; questa transazione viene definita **"root"** poiché è la transazione principale che conterrà tutte le altre sottotransazioni (quindi T_1, T_2, T_3, T_4, T_5).

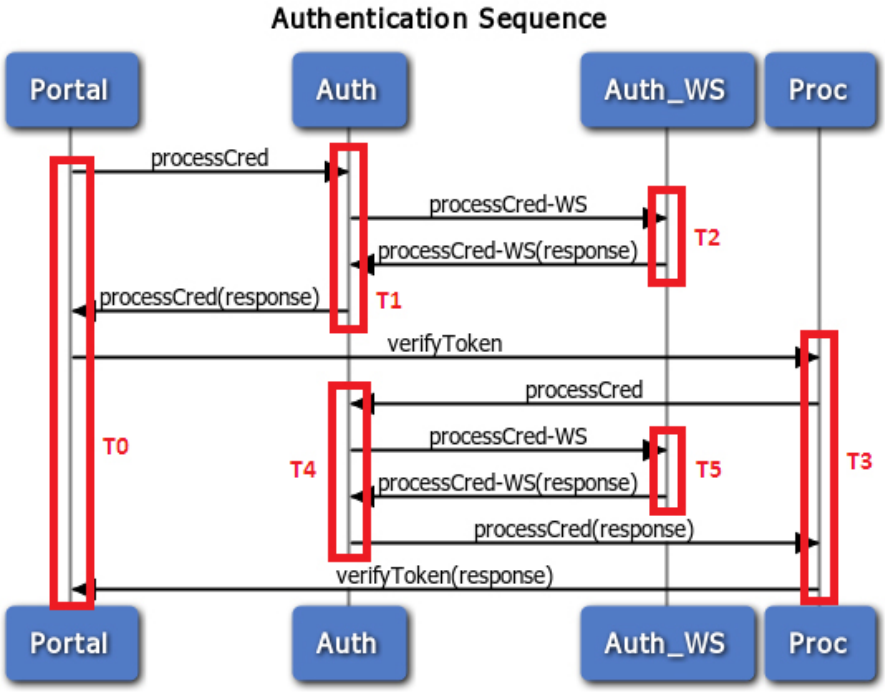


Figura 4.11: Sequence diagram della nostra transazione distribuita.

Portal appena riceve la stringa dal client invoca il processo BPEL Auth passandogli la stringa stessa. Auth è il componente chiave del nostro CBDS poiché invoca un Web Service esterno che svolge l'intera logica di business della transazione (quindi composto da codice Java). I processi BPEL si occupano infatti dell'orchestrazione del processo generale, passandosi i parametri e invocandosi nei momenti giusti, mentre il web service esterno di Auth (ricordiamo che il processo *Auth.bpel* e il web service esterno che invoca rappresentano un unico componente ai fini della gestione della version consistency) ha il compito di eseguirne la logica. Auth viene invocato sia da Portal che da Proc nel corso dell'intera transazione, quindi è previsto che, oltre al parametro di ingresso, gli venga passato come parametro anche il tipo di operazione da eseguire.

Tornando alla nostra transazione, Auth viene invocato da Portal che specificherà, grazie al parametro di ingresso, che dovrà essere essere eseguita l'operazione 1. Auth riceve la stringa e la tipologia di operazione e indirizza questi due parametri al proprio Web Service esterno Auth_WS, invocando il metodo *processCred-WS*. Il web service, ricevendo in ingresso la stringa (denominata *Cred*) e l'*operation*, eseguirà l'operazione di calcolo della lunghezza della stringa e restituirà questo valore (convertito in String)al processo BPEL Auth. Auth a sua volta restuituirà questo parametro a Portal.
A questo punto entra in gioco Proc che viene invocato da Portal per eseguire il metodo *verifyToken*: Proc invoca Auth con lo scopo di eseguire un controllo sulla

lunghezza della stringa ricevuta da Portal. Auth viene quindi istanziato nuovamente e dovrà invocare una seconda volta il proprio web service.
Questa volta gli passerà come parametri la stringa ricevuta da Proc (che ricordiamo rappresenta la lunghezza della stringa iniziale trasmessa dal client a Portal, calcolata dal web service durante la sua invocazione precedente *Auth_WS*) nel parametro di ingresso *Token* e come tipologia di *operation* gli passerà il valore 2. Auth_WS quindi convertirà la stringa ricevuta in un valore di tipo `Integer` ed eseguirà una logica che prevede di restituire ad Auth la stringa `true` se questo valore intero risulta maggiore o uguale di 2 oppure la stringa `false` se il valore intero è minore di 2.
Il valore 2 è casuale, ci interessava soltanto svolgere una certa logica per accettare o meno l'autenticazione che potesse essere successivamente modificata ai fini delle nostre richiste di aggiornamento a runtime.
La sequenza di passaggi di parametri si conclude con Auth_WS che passa la stringa risultante a Proc come risultato del metodo *processCred*; Proc che la restituisce a sua volta a Portal come risultato del metodo *verifyToken* e Portal che trasmette questo esito al client che ha avviato la transazione.

Per eseguire la transazione T_0 è quindi assolutamente necessario che vengano svolte le sottotransazioni T_1, T_2, T_3, T_4, T_5. L'ordine delle attività viene gestito dai processi BPEL e dalla loro coordinazione. Abbiamo scelto di utilizzare dei parametri di ingresso e di uscita (sia per i processi BPEL sia per i web services) di tipo `String`, ad eccezione del valore che identifica l'operazione da svolgere che è invece un `Integer`, poiché questa scelta ci creava meno problemi per gestire la fase di invocazione. Quindi la stringa che arriva direttamente dal client ad inizio transazione resta ovviamente una stringa, la sua lunghezza, che viene calcolata alla prima invocazione di Auth_WS, viene convertita da `Integer` a `String`, mentre l'esito dell'autenticazione dopo il controllo sulla lunghezza in questione potrebbe essere un `Boolean` ma invece viene passato sempre come `String`.

Riprendendo le definizioni riguardanti le transazioni presentate nel capitolo 2, è evidente che le transazioni eseguite dal nostro CBDS presentano le seguenti relazioni:

- $sub(T_0, T_1), sub(T_1, T_2), sub(T_0, T_3), sub(T_1, T_2), sub(T_3, T_4), sub(T_4, T_5)$
- $ext(T_0) = \{T_1, T_2, T_3, T_4, T_5\}$
- $ext(T_1) = \{T_2\}$
- $ext(T_2) = \{\}$
- $ext(T_3) = \{T_4, T_5\}$
- $ext(T_4) = \{T_5\}$
- $ext(T_5) = \{\}$

4.2.2 I componenti: definizione dei processi BPEL

Analizziamo in dettaglio i componenti BPEL del nostro sistema.
Portal.bpel è il processo che resta in attesa delle chiamate del client e che restituisce il risultato finale dell'intera transazione. Portal infatti esegue la root transaction T_0.
In figura 4.12 viene mostrato il design di questo processo BPEL, con il dettaglio delle attività e del loro flusso di esecuzione. Ogni processo BPEL può essere considerato comunque come un servizio da invocare, quindi proprio come nel caso dei web services, esporrà il proprio WSDL attraverso il quale vengono esposte le operazioni disponibili con i relativi parametri di ingresso e di uscita.

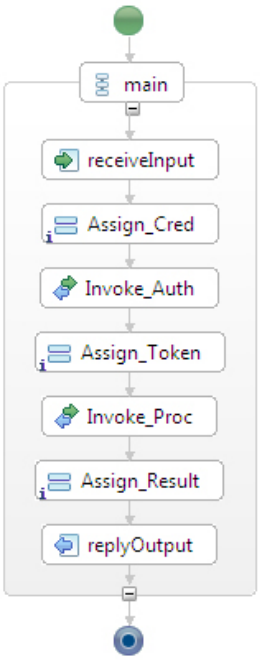


Figura 4.12: Rappresentazione del processo Portal.bpel.

Al momento dell'avvio dell'engine Active-Bpel su cui è stato fatto il suo deploy, *Portal.bpel* viene istanziato e si mette in attesa di chiamate dai client. Nel seguente elenco vengono spiegate dettagliatamente le attività eseguite in sequenza:

- ***receiveInput:*** all'istante della chiamata, la stringa ricevuta dal client viene memorizzata nella variabile *input*;

- ***Assign_Cred:*** durante questa attività di assegnamento, il valore della variabile *input* viene assegnato al parametro *cred* della variabile *auth_plRequest*,

che verrà utilizzata nell'attività successiva in ingresso all'invocazione di Auth. Inoltre viene assegnato il valore `1` al parametro *operation* della stessa variabile con più parametri;

- ***Invoke_Auth:*** dopo aver ricevuto la stringa dal client e averla memorizzata nell'apposito parametro, Portal, con questa attività, invoca il processo BPEL Auth. In ingresso passerà ad Auth la variabile *auth_plRequest*, appositamente preparata e valorizzata durante l'attività di <Assign> precedente. Questa variabile è un po' particolare poiché contiene al suo interno tre parametri: *cred* e *token* di tipo `String` e *operation* di tipo `Integer`;

- ***Assign_Token:*** a questo punto Portal ha ricevuto da Auth la lunghezza della stringa, che è stata memorizzata nella variabile *auth_plResponse* al momento della risposta di Auth. Quindi con questa attività di assegnamento copia il valore nella variabile *proc_plRequest* predisponendo la chiamata successiva a Proc;

- ***Invoke_Proc:*** viene invocato il processo BPEL Proc, che riceve come parametro in ingresso la lunghezza della stringa trasmessa dal client(contenuta nell'unico parametro *token* della variabile *proc_plRequest* valorizzata in precedenza;

- ***Assign_Result:*** Portal riceve l'esito del controllo di autenticazione (`true` o `false`) da Proc e assegna alla variabile *output* il valore della variabile *proc_plResponse* che contiene appunto questa stringa di esito. In questo modo Portal è pronto a restituire al client il risultato finale;

- ***replyOutput:*** con questa attività finale Portal trasmette al client (da cui era stato invocato) il risultato finale della transazione, che sarà `true` se il controllo di autenticazione è andato a buon fine o `false` in caso contrario.

É interessante notare che la comunicazione tra Portal ed Auth avviene in modo sincrono: all'interno di un'unica attività di <Invoke> viene effettuata la chiamata e si riceve anche la risposta. Invece la comunicazione tra Portal e il client avviene attraverso una comunicazione asincrona, essendoci in gioco varie attività in sequenza che devono essere eseguite prima della restituituzione del risultato. Quindi all'inizio del processo c'è un'attività di <Receive> per ricevere la chiamata e poi alla fine della transazione ci sarà un'attività di <Reply> per restituire il risultato al client.

Auth.bpel è il processo chiave dell'intera transazione poiché attraverso il web service ad esso associato è in grado di eseguire la logica di business che gestisce il meccanismo di autenticazione. Auth viene invocato sia da Portal che da Proc, e, a seconda dei parametri che riceve in ingresso esegue due operazioni differenti (pur esponendo un'unica operazione nel suo WSDL, che però è in grado di eseguire

in alternativa entrambe le funzionalità richieste). Come per la definizione in linguaggio BPEL di Portal, analizziamo le singole attività che lo compongono e il flusso di esecuzione mostrato in figura 4.13.

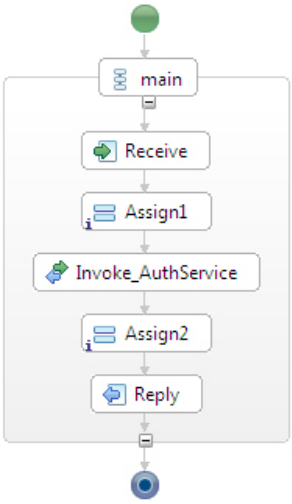


Figura 4.13: Rappresentazione del processo Auth.bpel.

Auth viene istanziato al momento della ricezione della chiamata (da parte di Portal o di Proc). L'attività di <Receive> è fondamentale poiché viene ricevuta in ingresso la variabile *authpl_Request* che contiene i parametri che permettono ad Auth di sapere quale operazione dovrà eseguire. I parametri contenuti nella variabile sono:

- ***Cred:*** viene valorizzato in caso di invocazione da parte di Portal e contiene il valore della stringa trasmessa dal client per l'autenticazione;
- ***Token:*** viene valorizzato in caso di invocazione da parte di Proc e contiene la lunghezza della stringa di autenticazione;
- ***Operation:*** viene valorizzata in entrambi i casi. Se Auth viene invocato da Portal avrà valore 1 perché sarà necessario calcolare la lunghezza della stringa di autenticazione, se invece Auth viene invocato da Proc avrà valore 2 perché si dovrà valutare l'esito dell'autenticazione in base alla lunghezza della stringa.

Analizziamo nel dettaglio le attività svolte da Auth:

- ***Receive:*** in questa attività si riceve la variabile di ingresso che contiene i parametri necessari per comprendere quale operazione dovrà effettuare il web service;

- ***Assign1:*** la variabile *auth_plRequest* ed i valori dei relativi parametri vengono copiati nella variabile *authservice_plRequest*, che avrà la stessa struttura e verrà passata in ingresso al web service di Auth;

- ***Invoke_AuthService:*** con questa invocazione sincrona Auth chiama il proprio web service che eseguirà la logica di business. Ricordiamo che nel caso il parametro di ingresso *operation* abbia valore `1`, il web service calcolerà la lunghezza della stringa di autenticazione mentre nel caso in cui abbia valore `2` valuterà se la lunghezza della stringa è maggiore o minore di un certo valore restituendo `true` o `false` a seconda dell'esito del controllo;

- ***Assign2:*** la variabile *authservice_plResponse*, ricevuta in risposta dal web service, viene copiata interamente nella variabile *auth_plResponse* (rispettando la tipologia dei parametri);

- ***Reply:*** Auth risponde al processo BPEL invocante (Portal o Proc) restituendogli la variabile *auth_plResponse* valorizzata nel corso dell'attività di assegnamento precedente. Sia Portal che Proc sapranno quali parametri di risposta utilizzare in base alla richiesta effettuata (Portal estrarrà dalla variabile la lunghezza della stringa di autenticazione contenuta nel parametro *Cred* mentre Proc estrarrà l'esito del controllo di autenticazione contenuto nel parametro *token*).

Proc viene invocato da Portal con lo scopo di valutare se la lunghezza della stringa ricevuta dal client è idonea o meno all'autenticazione. Per eseguire questo compito Proc invoca il processo Auth, già analizzato in precedenza. In figura 4.14 vengono mostrare le attività e il flusso di esecuzione di *Proc.bpel*.

Proc viene istanziato nell'istante in cui viene chiamato da Portal, e svolge le seguenti attività:

- ***Receive_Portal:*** riceve la chiamata di Portal, con la variabile di ingresso *proc_plRequest* contenente i tre parametri *Cred*, *Token* e *Operation* di cui si è discusso anche in precedenza. In questo caso sarà il parametro *Token* ad essere valorizzato, poiché Proc dovrà incaricarsi del controllo sulla lunghezza. Quindi *Token* conterrà il valore della lunghezza della stringa ricevuta da Portal, in formato String;

- ***Assign_Token:*** con questa attività di assegnamento Proc predispone la variabile *auth_plRequest* per la chiamata ad Auth, valorizzando il parametro *Token* ed impostando il parametro Operation con il valore `2`;

- ***Invoke_Auth:*** Proc invoca Auth passandogli in ingresso la variabile *auth_plRequest* e ricevendo in risposta la variabile *auth_plResponse* che conterrà un unico parametro con l'esito del controllo sulla lunghezza della stringa(*true* oppure *false*);

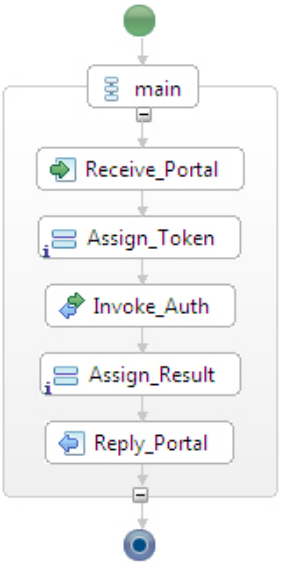


Figura 4.14: Rappresentazione del processo Proc.bpel.

- ***Assign_Result:*** durante questo assegnamento Proc valorizza l'unico parametro della variabile *proc_plResponse* con il valore del parametro *Token* della variabile *auth_plResponse*. In questo modo è pronto per comunicare il risultato finale della transazione a Portal;

- ***Reply_Portal:*** tramite questa attività di <Reply>, Proc comunica il risultato finale della transazione a Portal, che a sua volta lo comunicherà al client che ha effettuato la richiesta.

A livello tecnico, la definizione di un processo BPEL si basa su due file: un file *.bpel* che contiene la definizione vera e propria del processo con la specifica delle attività e del loro flusso di esecuzione, e un file *.pdd* in cui vengono descritti i vari PartnerLink del processo.
I *PartnerLink* sono dei canali di comunicazione con entità esterne, ad esempio con i processi BPEL partner.
All'interno di un PartnerLink deve essere specificato anche il *PartnerRole*, che è fondamentale in caso di invocazioni ad altri processi o a web service esterni, poiché contiene l'endpoint e il nome del servizio da invocare. In sostanza quindi il PartnerLink contiene la definizione dei partner da invocare.
Sia il file *.bpel* sia il file *.pdd* sono basati sul linguaggio XML. Per maggiori dettagli sull'installazione e l'uso dei processi di business si rimanda il lettore all'appendice B.

4.2.3 Il web service di business: Auth_WS

I processi BPEL che costituiscono i componenti del nostro sistema, servono, come detto, per orchestrare la transazione e per i passaggi di parametri; la logica di business vera e propria viene però svolta da un web service classico chiamato Auth_WS, che viene invocato dal processo Auth.
Come ricordato in precedenza, ai fini della gestione della Version Consistency Auth e Auth_WS sono considerati come un unico componente; quindi sia una modifica del processo BPEL sia una modifica del web service devono essere considerate come un aggiornamento del medesimo componente. Tuttavia non devono necessariamente risiedere sullo stesso server (infatti nel nostro caso il deploy è fatto su due server diversi).

La creazione del web service avviene scrivendo una classe Java integrata con delle apposite annotazioni (per dettagli consultare B).
Il tag *@WebService* indica semplicemente che quella classe verrà utilizzata come web service mentre il tag *@WebMethod* indica che un determinato metodo della classe costituirà una operazione esposta dal web service.
Nel nostro caso il web service Auth_WS esporrà un'unica operazione denominata *processCred*. Questo metodo riceverà in ingresso 3 parametri dal processo BPEL Auth: *Cred*, *Token* e *Operation*; il loro significato è già stato descritto ampiamente nel corso di questo capitolo, nel paragrafo in cui viene analizzata la transazione del nostro CBDS.
Descriviamo quindi soltanto brevemente la logica: se *Operation* vale `1` viene calcolata la lunghezza della stringa *Cred*, che è poi convertita in formato `String` e restituita ad Auth; se invece *Operation* vale `2`, la stringa *Token* viene convertita nel tipo `Integer` e si verifica se questo valore è maggiore o uguale a 2. Se così è, viene restituita ad Auth la stringa `true`, in caso negativo viene restituita la stringa `false`.

4.3 Il framework

4.3.1 Architettura del sistema

La nostra implementazione si basa sulla struttura sintetizzata in figura 4.15.
Ogni processo BPEL dell'intera transazione distribuita, ovvero Portal.bpel, Auth.bpel e Proc.bpel, per rappresentare un nodo di un sistema distribuito deve essere eseguito su macchine diverse; per questo motivo ci siamo dovuti servire di tre installazioni Active-Bpel ognuna indipendente dall'altra e eseguita su una istanza dedicata del container Tomcat.
La gestione degli aggiornamenti attraverso il criterio di *Version Consistency* viene realizzata sfruttando due elementi principali: Dynamo e il web service VersionConsistencyService che contiene la logica di business per l'esecuzione dell'algoritmo e

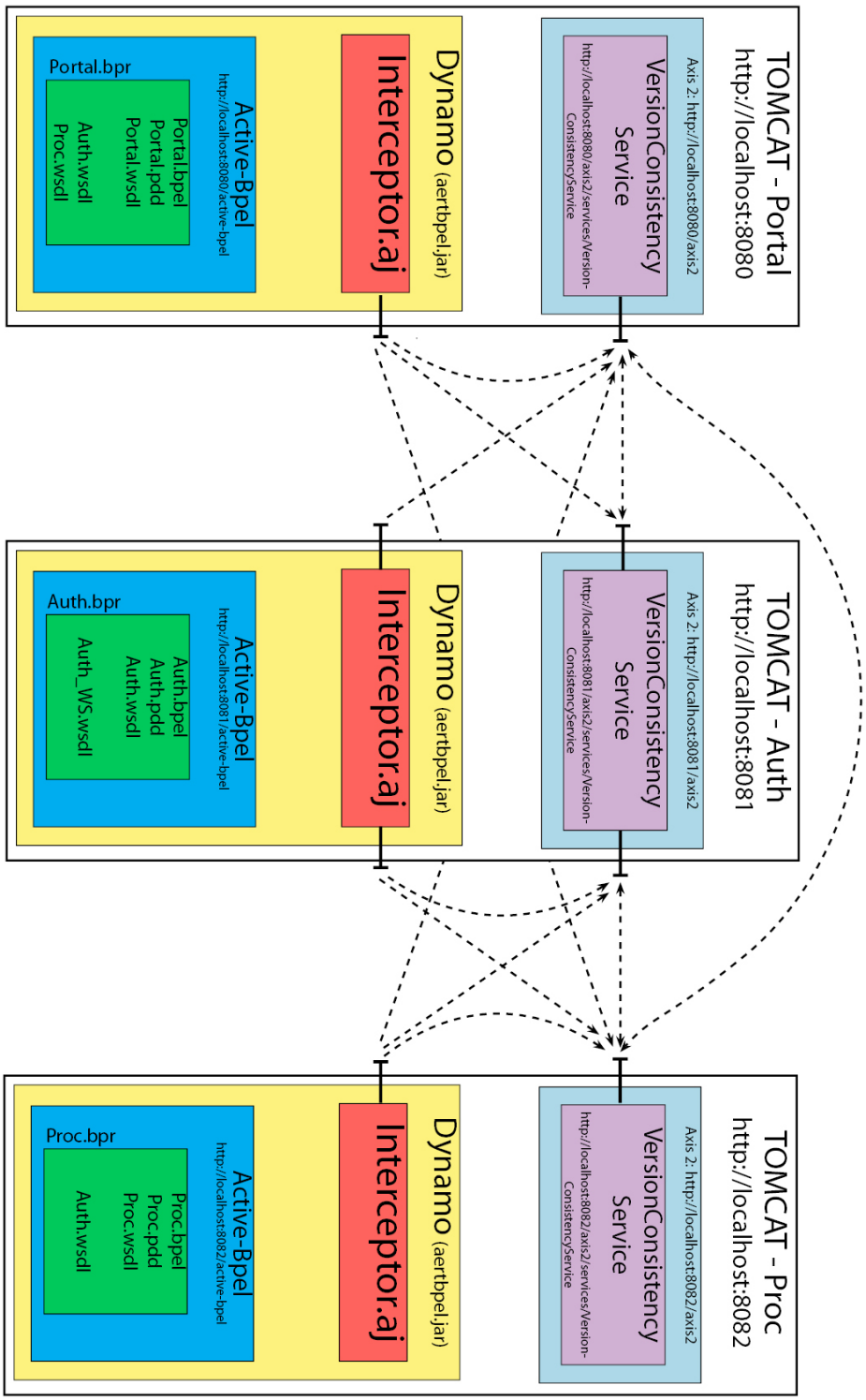


Figura 4.15: Architettura del nostro sistema distribuito.

la gestione degli update dinamici dei componenti.
Dynamo, come affermato in precedenza, estende le funzionalità dell'engine Active-bpel consentendo di intercettare l'esecuzione di un processo BPEL in determinati istanti temporali, che precedono o succedono all'esecuzione di alcune attività del processo. Rappresenta quindi uno strumento ideale per eseguire nei momenti giusti la logica di business per la gestione degli aggiornamenti.
L'esecuzione dei processi BPEL viene intercettata nei seguenti punti di interesse:

- prima dell'inizio del processo (punto di intercettazione before start process);
- prima di un'attività di <receive> (punto di intercettazione before receive);
- prima di un'attività di <invoke> (punto di intercettazione before invoke);
- dopo di un'attività di <invoke> (punto di intercettazione after invoke);
- prima di un'attività di <reply> (punto di intercettazione before reply);
- dopo di un'attività di <reply> (punto di intercettazione after reply);

Durante questi punti di intercettazione vengono invocati gli appositi metodi del proprio web service (o di altri web service, con alcune limitazioni di cui discuteremo) per la gestione dell'algoritmo: la logica di business del criterio di Version Consistency è contenuta interamente nei VersionConsistency web service. Ogni processo BPEL del nostro modello di CBDS deve essere associato al proprio Dynamo, per questo Portal, Auth e Proc vengono intercettati a runtime dalle proprie istanze di Dynamo, installate su ciascuno dei tre Tomcat.
Entrando nei dettagli tecnici, Dynamo non è nient'altro che una libreria che aggiungiamo ai nostri server Tomcat, sovrascrivendone una già esistente chiamata *ae_rtbpel.jar*; così facendo abbiamo aggiunto del codice Java ad Active-bpel, attraverso la classe *Interceptor.aj*.
Questa è una normale classe Java estesa con le funzionalità del linguaggio *AspectJ*, il quale permette di intercettare i processi in esecuzione nei punti giusti. Inoltre l'interceptor sfrutta le librerie *Axis* che ci permettono di invocare i metodi dei web service di gestione della Version Consistency. L'engine di esecuzione viene quindi integrato con le nuove funzionalità necessarie all'esecuzione degli aggiornamenti dinamici.

La gestione dell'algoritmo di Version Consistency viene svolta da un web service esterno che contiene la struttura dati necessaria e tutti i metodi che compongono la logica di business; questo servizio è chiamato VersionConsistencyService. Ogni processo BPEL che fa parte della transazione deve avere la propria istanza del web service della Version Consistency e per questo motivo la nostra architettura prevede il deploy di una sua istanza su ciascun Tomcat. A differenza del web service invocato dal componente Auth (presentato nei paragrafi precedenti)

che si occupa dell'esecuzione della logica di business della transazione, il servizio VersionConsistencyService sfrutta le librerie *Apache Axis2*; inoltre, questo web service è di tipo ***stateful***, ovvero mantiene in memoria l'istanza delle strutture dati su più richieste: questo si traduce nel fatto che i dati sugli archi dinamici sono mantenuti all'interno del web service e dunque non è necessario avere un database dedicato.
Per maggiori dettagli tecnici sul framework è possibile consultare l'appendice C.

In figura 4.15 sono mostrate anche le possibili interazioni tra i vari componenti che gestiscono la Version Consistency. Per interazione ovviamente si intende l'esecuzione di una chiamata ad un metodo esposto dai VersionConsistency web services.
Ciò rappresenta un punto chiave perché viene evidenziato il concetto di conoscenza locale previsto dal criterio di Version Consistency. Come evidenziato nel capitolo 3, l'algoritmo di Version Consistency ha un grande punto di forza: un componente (nodo) del sistema distribuito può verificare, sfruttando solamente la sua conoscenza locale, se è in stato di *free* o meno per l'esecuzione di un corretto aggiornamento. Ogni nodo possiede solo i propri archi della configurazione dinamica e conosce soltanto i componenti del sistema con cui avrà un'interazione. Quindi anche all'interno della nostra implementazione queste due caratteristiche vengono mantenute: l'interceptor conosce soltanto gli archi relativi al proprio processo BPEL, poiché sono memorizzati nel VersionConsistencyService ed inoltre ogni interceptor potrà invocare soltanto il suo web service o quello dei componenti "vicini" nel grafo di configurazione statica.
All'interno del nostro modello di CBDS i tre componenti sono tutti connessi tra di loro staticamente, quindi ogni interceptor avrà la possibilità di invocare tutti i web services dei tre componenti, ma in caso di utilizzo di numerosi componenti che non siano tutti connessi staticamente tra loro, la nostra implementazione sarebbe comunque in grado di gestire l'algoritmo basandosi su conoscenza locale e non globale.

Riassumendo, un interceptor richiede funzionalità al VersionConsistencyService locale e anche a quelli esterni sotto forma di SOAP requests; i VersionConsistencyService rispondono restituendo le relative SOAP responses. Non succede mai il contrario, ovvero i VersionConsistencyService non richiedono mai servizio agli interceptor, poichè l'interceptor non è un web service; in più i VersionConsistencyService possono anche comunicare tra di loro, purchè siano collegati staticamente.

4.3.2 Implementazione della configurazione statica

Ogni componente del sistema ovviamente avrà la necessità di conoscere alcuni parametri statici, come ad esempio gli url dei web service da invocare o i path dei file BPEL da cui leggere alcune specifiche. Questo si traduce nella mappa-

tura della configurazione statica necessaria all'algoritmo di Version Consistency all'interno della nostra soluzione: l'utente dovrà personalizzare sia l'interceptor di Dynamo sia il web service impostando direttamente all'inizio delle relative classi Java alcuni parametri specifici di ogni processo BPEL. Analizziamo nel dettaglio ciò che è necessario configurare (su ogni componente) per una corretta esecuzione della nostra implementazione:

- **Interceptor di Dynamo**:
  - Parametro **rootTrans**: stringa che contiene il nome della root transaction. Tutti i componenti del sistema che eseguono la transazione generale devono infatti conoscere qual è la *root transaction* (ovvero il processo che riceve la richiesta dal client e avvia la transazione, nel nostro caso *Portal.bpel*);
  - Parametro **myName**: stringa che contiene il proprio nome del processo. Ad esempio nell'interceptor di Portal bisognerà impostare `Portal.bpel`;
  - Parametro **endPointRef**: stringa che contiene l'url del proprio web service VersionConsistencyService, i cui metodi verranno invocati nei vari punti di intercettazione durante l'esecuzione del processo. Ad esempio nel nostro caso per quanto riguarda Portal.bpel l'url sarà: `http://localhost:8080/axis2/services/VersionConsistencyService`;
  - Parametro **namespaceUri**: stringa che contiene il namespace di Axis2 indicato nel proprio sistema. Nel nostro caso è il seguente: `http://ws.apache.org/axis2`.
- **VersionConsistencyService**:
  - parametro **bpel_path**: stringa che contiene il path del file *.bpel* che definisce il processo, quindi ogni componente ovviamente avrà un path diverso. Nel nostro caso ad esempio sul VersionConsistency web service di Portal.bpel abbiamo impostato il seguente path: `C:/tomcat5/tomcat_portal/bpr/work/ae_temp_New_Portal_bpr/bpel/New_Portal/bpelContent/`;
  - Parametro **pdd_path**: stringa che contiene il path del file *.pdd* associato al file *.bpel*, contenente le dichiarazioni dei processi partner. Sul VersionConsistencyService di Portal.bpel abbiamo impostato il seguente path:`C:/tomcat5/tomcat_portal/bpr/work/ae_temp_New_Portal_bpr/META-INF/pdd/New_Portal/bpelContent/`;
  - Parametro **bpel_root**: come avviene nell'interceptor, in questa stringa viene dichiarato il nome della transazione di root;

- Parametro **myName**: stringa che contiene il nome del proprio processo, esattamente come nei parametri dell'interceptor descritti in precedenza;
- Parametro **linkMap**: è un hashmap che serve per associare al nome di un processo l'indirizzo del corrispondente VersionConsistencyService. Quindi i nomi dei processi fungeranno da chiavi e gli URL saranno i valori ad essi associati. Sia le chiavi che i loro valori sono definiti a loro volta come parametri (di tipologia String). L'hashmap viene utilizzato per conoscere gli URL ed invocare così i web service dei processi connessi staticamente al nodo corrente. É importante evidenziare che un componente, per rispettare tutto ciò che è stato dichiarato fino ad ora, deve avere conoscenza locale quindi l'hashmap non potrà contenere nomi di processi e corrispondenti url di web service non connessi staticamente (in ingresso o in uscita) al componente stesso. Con l'analisi del paragrafo precedente abbiamo già chiarito che nella nostra configurazione statica d'esempio tutti i componenti (Portal,Auth e Proc) sono connessi staticamente tra loro quindi il nostro hashmap in tutti i web service VersionConsistencyService conterrà le seguenti associazioni:
  - Portal.bpel -> `http://localhost:8080/axis2/services/VersionConsistencyService`
  - Auth.bpel -> `http://localhost:8081/axis2/services/VersionConsistencyService`
  - Proc.bpel -> `http://localhost:8082/axis2/services/VersionConsistencyService`

  Gli indirizzi dei nostri web service evidenziano chiaramente che ognuno di essi è deployato su un'istanza differente di Tomcat. Infatti per simulare l'esecuzione delle transazioni su un sistema distribuito i server vengono avviati tutti nel nostro ambiente locale ma su porte diverse(80,81 e 82).
- Parametro **oldEpr**: per effettuare un aggiornamento dinamico nel nostro sistema cambieremo l'indirizzo del web service esterno associato ad Auth che esegue la logica di business della transazione. Descriveremo nel dettaglio questa operazione nei paragrafi successivi, per il momento è sufficiente comprendere che questo parametro di configurazione contiene in una stringa l'indirizzo della prima versione (ovvero quella che verrà aggiornata) del web service esterno che viene invocato da Auth. Nel nostro caso di esempio il parametro è impostato quindi a `http://localhost:8080/axis/Auth_Service.jws`. Nella classe Java dei VersionConsistencyService associati a componenti che non prevedono alcun aggiornamento questo parametro deve essere lasciato vuoto.

– Parametro **newEpr**: come sopra, contiene l'url della nuova versione del web service invocato da Auth.
– Parametro **namespaceUri**: stesso parametro presente anche nell'interceptor di Dynamo, che indica il namespace utilizzato da Axis2 sul proprio sistema.

4.3.3 Struttura dati della configurazione dinamica

Il nostro web service VersionConsistencyService gestisce l'esecuzione dell'algoritmo presentato nel capitolo 3 e garantisce la corretta esecuzione degli aggiornamenti dinamici sui componenti dei sistema. Per raggiungere questo obiettivo ovviamente utilizza i numerosi metodi che compongono la logica di business ma il fatto più rilevante è che il web service deve essere di tipo *stateful*, ovvero deve avere una struttura dati ben definita persistente tra le varie chiamate provenienti dall'interceptor o da altre istanze del web service stesso.
Questa struttura dati memorizzerà la configurazione dinamica del sistema per ogni componente. La conoscenza è locale, quindi ogni componente del sistema attraverso il proprio VersionConsistencyService conoscerà solamente i propri archi, in ingresso e in uscita, ignorando gli archi appartenenti agli altri membri del sistema. É infatti sufficiente una conoscenza locale di questo tipo per far sì che un web service analizzi se è possibile aggiornare o meno il proprio componente. Riprendiamo in considerazione la transazione presentata nel capitolo 3 ad un istante di tempo qualsiasi, come mostrato in figura 4.16.

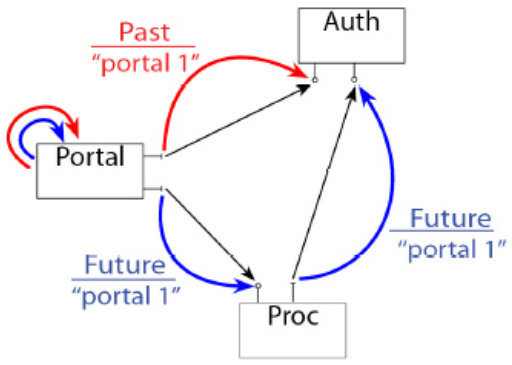


Figura 4.16: Struttura generale dell'implementazione realizzata.

Gli archi che collegano i vari componenti rappresentano la configurazione dinamica, quindi un obiettivo fondamentale della struttura dati del nostro web service consiste nel modellizzare e nel mantenere in memoria questi archi; ogni componente del sistema manterrà quindi memorizzati i suoi archi nel proprio

VersionConsistencyService. Il web service sarà in grado di verificare, in base agli archi presenti nella configurazione dinamica, se il componente può essere aggiornato in modo sicuro o meno.
La nostra scelta implementativa è stata quella di realizzare una sottoclasse che modellizzasse gli archi, contenuta direttamente nella classe Java del VersionConsistencyService. Questa sottoclasse si chiama *DynEdges_Struct* e conterrà le seguenti quattro variabili di tipologia String che sono sufficienti a modellizzare in modo preciso un arco:

- **source:** identifica il nodo sorgente dell'arco (quindi un processo .bpel);
- **dest:** identifica il nodo destinazione dell'arco (quindi anche in questo caso un processo .bpel);
- **time:** identifica il tempo della transazione, ovvero `future` per le invocazioni che potranno avvenire in futuro o `past` per le invocazioni già avvenute (e terminate) in precedenza;
- **transaction:** identifica la transazione di root. Gli archi della configurazione dinamica possono infatti appartenere a transazioni differenti e ai fini dei controlli per verificare la *freeness* di un nodo è fondamentale sapere a quale transazione di root appartengono.

Gli archi della configurazione dinamica sono quindi oggetti (o se preferite istanze) della classe DynEdges_Struct e la configurazione dinamica totale, che è quindi un insieme di questi archi, viene modellizzata attraverso un ArrayList di questi oggetti. Ogni web service conterrà due ArrayList di archi (quindi di oggetti di tipo <DynEdges_Struct>),uno per l'insieme degli archi in ingresso (chiamato dyn_in) e uno per l'insieme degli archi in uscita (chiamato dyn_out). Grazie a questi due array qualsiasi componente tramite il web service ad esso associato potrà conoscere in ogni momento i propri archi.

Oltre all'aspetto fondamentale della memorizzazione degli archi della configurazione dinamica, il web service necessita di numerosi altri dati per la corretta esecuzione della propria logica. Il ruolo di alcuni di essi sarà più chiaro nei prossimi paragrafi ma è comunque utile analizzarli con la seguente panoramica generale:

- **invoker_name:** ogni volta che durante la transazione viene effettuata un'invocazione tra processi, il processo invocante chiama un apposito metodo del web service del processo invocato e viene memorizzato in questa variabile il nome dell'invocante. Ciò è fondamentale perché altrimenti non ci sarebbe modo nei vari Dynamo o nei web service di conoscere il nome esatto del processo da cui si è stati invocati;

- **actual_pid:** ogni volta che durante la transazione viene effettuata un'invocazione tra processi, il processo invocante, come nel caso precedente, chiama un apposito metodo del web service del processo invocato e vi memorizza anche l'id del processo di root della transazione (più avanti analizzeremo il metodo utilizzato per far sì che ogni processo conosca sempre questo rootPid);

- **position:** in fase di inizializzazione, in base al file .bpel di ogni processo viene creato un apposito file *.xml* che elenca le invocazioni che verranno effettuate dal processo stesso durante l'esecuzione di una transazione. Questa variabile è un array che serve per mantenere in memoria la posizione sul file *.xml* per quanto riguarda le transazioni in corso, quindi è utile per stabilire i processi che vengono di volta in volta invocati in un determinato istante;

- **lock_pid:** questa variabile può assumere i valori `true` o `false` e serve per effettuare dei lock. In pratica viene utilizzata per fare in modo che un processo non possa sovrascrivere alcune variabili della struttura dati mentre un altro processo le sta utilizzando o le sta scrivendo a sua volta;

- **transactionsEprOld:** questo ArrayList di stringhe contiene gli id dei processi che vengono eseguiti con la vecchia versione di un componente in caso di aggioramenti; è un dato fondamentale in caso di aggiornamento con transazioni concorrenti;

- **transactionsEprNew:** questo ArrayList di stringhe contiene invece gli id dei processi che vengono eseguiti con la nuova versione di un componente in caso di aggioramenti; è un dato fondamentale in caso di aggiornamento con transazioni concorrenti;

- **currentVersionEpr:** nei web service dei componenti che prevedono aggiornamenti, questa variabile serve per identificare l'URL del servizio da invocare per le nuove transazioni. Quindi, analizzando il nostro caso di esempio concreto, quando viene aggiornato il web service di Auth, l'URL corrisponderà alla nuova versione del servizio(`newEpr`). Prima dell'aggiornamento invece questa variabile conterrà l'URL della versione iniziale del servizio(`oldEpr`);

- **isFree:** questa variabile è fondamentale perché viene scritta in seguito ai controlli sulla *freeness* di un nodo. Se assume valore `true` significa che il componente può essere aggiornato, altrimenti se assume il valore `false` significa che in base al criterio di Version Consistency il componente non può essere aggiornato;

- **rootPid_hash:** è un ArrayList di stringhe fondamentale per far sì che ogni processo conosca in ogni momento della sua esecuzione il process id della root transaction corrente e il nome del proprio processo invocante.

Inseriamo (in figura 4.17), a scopo di completezza, il class diagram che riassume la struttura dati del VersionConsistencyService contenente anche i metodi esposti dal web service utilizzati dall'interceptor (descritti nei dettagli nei prossimi paragrafi).

4.3.4 Le fasi dell'algoritmo

4.3.4.1 Set up

La fase di inizializzazione dell'algoritmo di Version Consistency prevede che vengano creati tutti gli archi `Future` della configurazione dinamica prima dell'inizio di una transazione distribuita. Come mostrato in precedenza, questi archi indicano che in futuro il nodo di destinazione dell'arco potrà essere invocato dal nodo sorgente. Ovviamente non è possibile che una coppia di componenti venga connessa con un arco `Future` se essi non sono collegati tra loro nella configurazione statica.

Nella nostra implementazione, per svolgere una corretta inizializzazione viene sfruttato il punto di intercettazione *Before start process*, che intercetta l'esecuzione di un processo nell'istante precedente l'avvio vero e proprio delle attività che lo compongono. Dovendo avviare questa fase dell'algoritmo prima che inizino le attività della transazione completa, all'interno del nostro sistema l'intercettazione avverrà poco prima dell'inizio della root transaction di Portal.bpel (che successivamente genererà tutte le altre sottotransazioni eseguite dagli altri componenti). I processi Auth.bpel e Proc.bpel quindi, al momento dell'inizializzazione, non essendo ancora stati invocati, non saranno istanziati. Tuttavia i loro VersionConsistencyService sono sempre attivi ed invocabili, ed è proprio su di essi che verranno memorizzati gli archi iniziali della configurazione dinamica.
I flussi di esecuzione e la successione delle attività dei vari processi BPEL che compongono il sistema sono disponibili grazie ai file *.bpel* che definiscono i processi stessi, quindi per la loro conoscenza non è necessario che un processo sia in esecuzione e nemmeno che sia istanziato. Grazie ai file *.pdd* che definiscono i partner, in caso di attività di <Invoke> è possibile ricavare anche il processo partner che verrà invocato. In fase di inizializzazione, ovvero nel *Before Start Process* della root transaction, è quindi possibile conoscere per ogni processo la sequenza dei processi partner che dovrà invocare per eseguire la propria sottotransazione. Ad esempio Portal.bpel dovrà invocare Auth.bpel e Proc.bpel, Proc.bpel dovrà invocare Auth.bpel e così via.

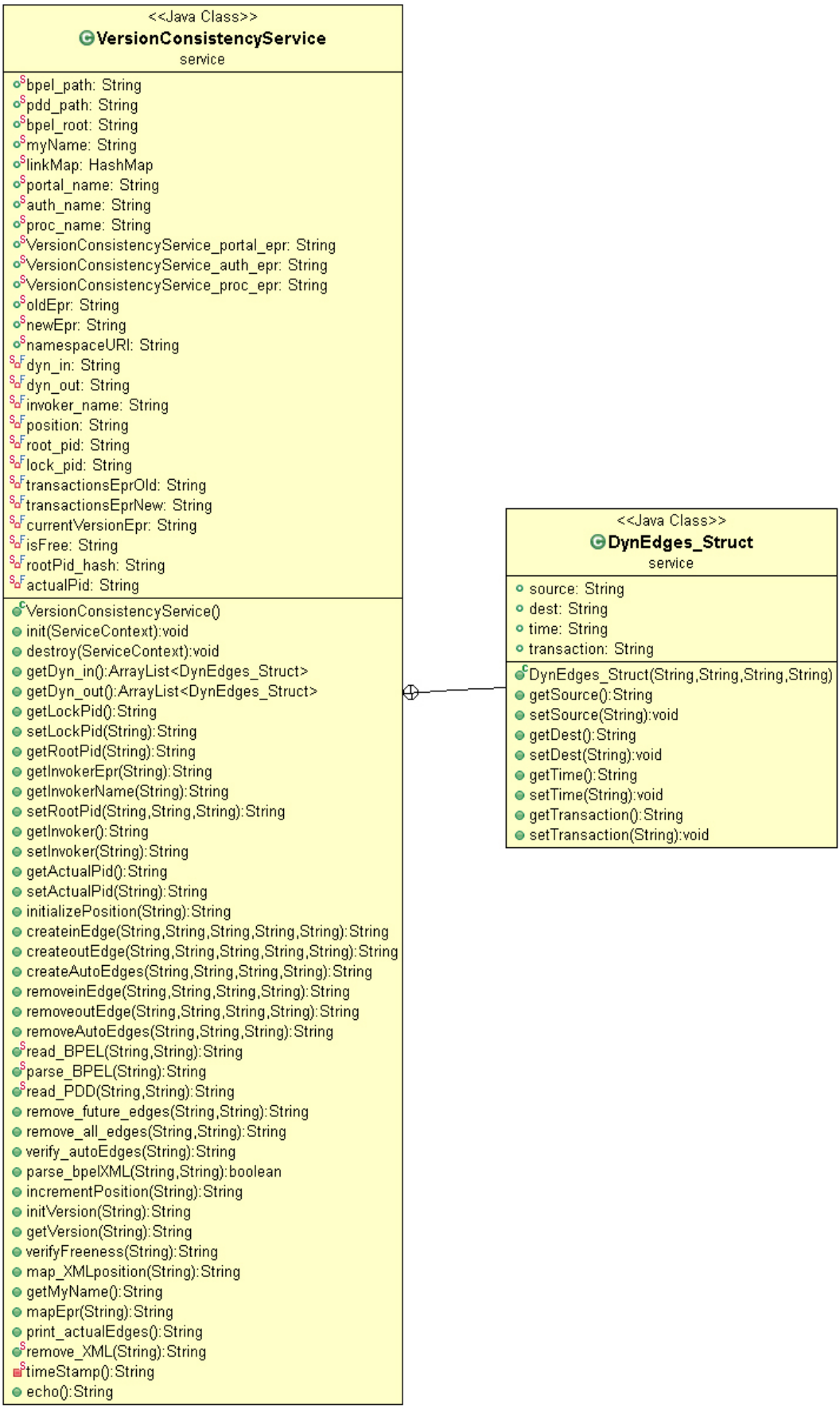


Figura 4.17: Class diagram del VersionConsistencyService.

Durante la fase di inizializzazione, che avviene prima dell'inizio di ogni transazione completa del sistema, per ogni processo BPEL viene creato un apposito file XML che conterrà la sequenza delle chiamate che verranno effettuate dal processo stesso. Questo file sarà fondamentale per tenere traccia della posizione durante l'esecuzione della transazione e durante la fase di progress dell'algoritmo. Senza questo file XML non sarebbe possibile sapere in un determinato istante quali processi sono stati invocati e quali dovranno esserlo in futuro. Ciò ovviamente è fondamentale anche per alcuni controlli dell'algoritmo, visto che è prevista l'eliminazione di determinati archi `Future` quando si ha la certezza che un processo partner non verrà più invocato. Il file XML viene utilizzato per uno di questi controlli, poiché è sufficiente scorrerlo dalla posizione corrente fino al termine per sapere quali processi BPEL devono ancora essere utilizzati. Di seguito viene mostrato il semplicissimo file XML di Portal.bpel:

```
<?xml version="1.0" encoding="UTF-8"?>
<sequence>
     <node>Auth.bpel<\node>
     <node>Proc.bpel<\node>
<\sequence>
```

Entrando maggiormente nei dettagli tecnici, la fase di inizializzazione all'interno del nostro sistema viene gestita con delle chiamate ricorsive a due metodi fondamentali : *read_Bpel* e *read_PDD*.
Nel punto di intercettazione *Before Start Process* dell'interceptor è presente un controllo per verificare se il processo che si sta intercettando esegue la root transaction. Soltanto in quel caso viene eseguita la fase di inizializzazione. In questo modo, quando i processi che eseguono sotto-transazioni (e non la root transaction) verranno istanziati, ovviamente non la eseguiranno di nuovo.
L'interceptor della root transaction invoca il metodo *read_Bpel* del proprio web service; questo metodo prima di tutto crea l'apposito file XML che memorizzerà la sequenza di chiamate e successivamente, sfruttando il path impostato dall'utente nei parametri della configurazione statica, effettua il parsing del file *.bpel* di definizione del processo.
Attraverso il parsing si analizza la sequenza delle attività che verranno svolte ed in caso di <Invoke> viene chiamato il metodo *read_PDD* che effettua a sua volta un parsing. Questa volta viene analizzato il file *.pdd* di definizione dei partner associato al processo *.bpel*; in questo modo si riesce a ricavare un'informazione fondamentale: il nome del processo che verrà invocato con l'attività di <Invoke> in questione.
Queste due informazioni vengono utilizzate per aggiornare il proprio file XML aggiungendo per ogni attività di <Invoke> un apposito tag che ha come contenuto il nome del processo da invocare. Prima di procedere con il parsing delle altre attività del processo BPEL vengono svolte altre due attività fondamentali:

1. vengono creati gli archi `Future` della configurazione dinamica, visto che dall'analisi effettuata si sa con certezza che per il corretto svolgimento della

transazione il processo di cui si sta effettuando il parsing invocherà il processo definito come partner nellattività di <Invoke>. Per cui viene creato un arco `Future` in uscita sul proprio web service e un arco `Future` identico in entrata nel web service del processo partner;

2. viene invocato il metodo *read_Bpel* sul web service del processo partner dell'attività di <Invoke>. In questo modo si procederà con il parsing completo del suo file *.bpel* e con le apposite chiamate al metodo *read_PDD* per ricavare anche i suoi partner, si genererà il suo file XML e si creeranno i suoi archi della configurazione dinamica.

Effettuando su tutti i processi queste chiamate agli stessi due metodi in modo ricorsivo al momento delle attività di <Invoke>, al termine della fase di inizializzazione tutti i componenti possiederanno:

- il proprio file XML con la sequenza dei processi partner che verranno invocati per l'esecuzione della transazione;
- gli archi della configurazione dinamica (uscenti se in futuro si invocherà un processo partner o entranti se si verrà invocati come partner da un altro processo) sul proprio web service VersionConsistencyService.

É importante mettere in evidenza che ogni processo possiede unicamente conoscenza locale. Conosce soltanto i processi che dovrà invocare o da cui verrà invocato, ma non conosce l'intera sequenza di invocazioni che verranno effettuate per l'esecuzione della transazione. Infatti durante la fase di inizializzazione per ogni componente è stato analizzato il proprio file *.bpel*, non si ha conoscenza di quello degli altri. Inoltre un componente (ad eccezione del processo che esegue la root transaction che avvia il tutto) esegue la propria fase di inizializzazione soltanto se viene invocato da un altro processo, e ciò significa che possiede un arco `Future` in ingresso e che di conseguenza saprà da chi viene invocato per essere parte della transazione completa.

4.3.4.2 Progress

Il punto cruciale dell'algoritmo di Version Consistency, e di conseguenza della nostra implementazione, è lo svolgimento della fase di progress.
Qui, nel corso delle varie transazioni vengono effettuati i controlli per stabilire se un nodo del sistema è aggiornabile o meno in un determinato istante.
Per quanto riguarda la configurazione dinamica invece vengono rimossi di volta in volta gli archi `Future` che non sono più necessari e vengono aggiunti gli archi `Past` quando un componente riceve la risposta da un altro componente da esso invocato (che quindi ha terminato la sua sottotransazione).
Gli aspetti tecnici principali progettati per questa fase (gestione del rootpid,

esecuzione dei controlli e degli aggiornamenti dinamici, rimozione degli archi nella configurazione dinamica) sono presentati nei paragrafi successivi. Qui verrà elencato in modo schematico ciò che avviene in ogni punto di intercettazione.
Before start process Ricordiamo che in questo punto di intercettazione, l'interceptor della root transaction avvia la fase di inizializzazione dell'algoritmo. Nella fase di progress invece i Dynamo dei componenti che non svolgono la root transaction (quindi vengono invocati da altri processi) eseguono le seguenti attività:

- ***getInvoker*** *(sul proprio web service)*: si ricava il nome del processo da cui si è stati invocati, che è stato memorizzato sul web service precedentemente all'attività di <Invoke> dal processo invocante stesso;

- ***getActualPid*** *(sul proprio web service)*: si ricava il process id della root transaction, che è stato precedentemente memorizzato sul web service dall'attività di <Invoke> del processo invocante. L'id viene quindi propagato tra tutti i processi ad ogni attività di <Invoke>;

- ***setRootPid*** *(sul proprio web service)*: viene creata una stringa concatenando il process id della root transaction, il nome dell'invoker e il proprio process id (ovviamente intervallati da un apposito spaziatore) in modo che in ogni punto di intercettazione il processo possa ricavare queste informazioni, che altrimenti verrebbero perse o sovrascritte se fossero memorizzate direttamente all'interno dell'interceptor;

- ***setLockPid*** *(sul proprio web service)*: viene impostato il lock a `false`, in modo che altri partner che invocano il processo possano scrivere sul web service del processo stesso il proprio nome (nome invoker) e il process id della root transaction. Il lock è fondamentale per gestire la concorrenza, altrimenti le variabili del web service che contengono le informazioni potrebbero essere sovrascritte prima che il processo memorizzi la stringa concatenata del punto precedente attraverso il metodo *setRootPid*;

- ***getRootPid*** *(sul proprio web service)*: il processo si fa comunicare dal proprio web service il pid della root transaction in previsione dell'attività successiva;

- ***initializePosition***: in fase di inizializzazione, come mostrato in precedenza, viene creato un file XML in cui viene memorizzata la sequenza dei partner che verranno invocati da un processo. Per gestire transazioni concorrenti, la posizione corrente sul file XML viene memorizzata in un apposito array. Ogni transazione che viene svolta da un processo sarà in una posizione differente in un determinato istante di tempo. Quindi attraverso questo metodo eseguito ad inizio transazione (nel Before Start Process) viene inizializzato

lo slot dell'array in cui verrà memorizzata la posizione corrente durante la transazione.

Before receive

- ***getRootPid*** *(sul proprio web service)*: all'inizio del punto di intercettazione si ricava dal proprio web service il process id della root transaction,che verrà utilizzato come parametro nei metodi successivi;

- ***createAutoEdges***: attraverso questo metodo vengono creati gli autoanelli che contraddistinguono la transazione in corso. Quindi verranno creati due anelli entranti e due uscenti (che rappresentano gli stessi archi, perché ovviamente nel caso di autoanelli un arco entrante in un nodo è anche arco uscente): un `Future` e un `Past` per ogni coppia.

Se l'interceptor non sta intercettando il processo che esegue la root transaction:

- ***getInvokerEpr*** *(sul proprio web service)*: si ricava il nome del processo da cui si è stati invocati e tramite l'hashmap che rappresenta la configurazione statica (configurato come parametro dall'utente) si ricava l'endpoint reference del suo web service;

- ***remove_future_edges*** *(sul web service del processo invocante)*: viene invocato il metodo, che ha il compito di controllare se è possibile rimuovere archi `Future` nella configurazione dinamica in uscita del processo invocante. In caso di eliminazione verranno effettuate chiamate ricorsive sui nodi destinazione degli archi eliminati;

- ***setLockPid*** *(sul proprio web service)*: questa chiamata non è sempre necessaria. Viene sfruttata quando nella definizione del processo BPEL intercettato sono presenti dei Correlation Set, ovvero degli elementi che fanno sì che il processo possa essere eseguito più volte senza essere reistanziato (a volte proseguendo la sua esecuzione da un'attività intermedia e non dall'attività iniziale). In questi casi è necessario impostare il lock a `false` perché questa operazione non viene svolta nel punto di intercettazione Before Start Process durante l'esecuzione delle transazioni che non riprendono dall'attività iniziale.

Before invoke Se il processo in esecuzione non svolge la root transaction:

- ***getRootPid*** *(sul proprio web service)*: all'inizio del punto di intercettazione si ricava il process id della root transaction.

In tutti gli interceptor (sia nell'interceptor del processo che esegue la root transaction sia negli interceptor dei processi che eseguono la sottotransazione):

- ***map_XMLposition*** *(sul proprio web service)*: viene letta la posizione corrente nell'apposito slot dell'array delle posizioni e sfruttando questa informazione si verifica sul file XML quale sarà il nome del processo che verrà invocato con l'attività di <Invoke>. A questo punto il webservice cerca l'endpoint reference associato al nome del processo ricavato (quindi questo epr è praticamente l'URL del web service del processo partner invocato) e lo comunica al processo. Tutte le attività dei punti successivi in questo punto di intercettazione verranno svolte soltanto se l'endpoint non è nullo, ovvero se il web service associato al processo partner è effettivamente esistente;

- ***getLockPid*** *(sul web service del processo invocato)*: si verifica sul web service del processo da invocare se il lock è impostato a `true`. In caso positivo, significa che qualche altro processo sta già scrivendo le variabili che identificano il nome dell'invoker e il process id della root transaction che viene propagato; in questo caso si attende che il lock venga impostato a `false`. Tutto ciò serve per gestire le transazioni concorrenti;

- ***setLockPid*** *(sul web service del processo invocato)*: quando viene verificato che il web service del processo partner non è bloccato dal lock, il processo invocante lo setta impostando l'apposito parametro a `true` per poter comunicare le informazioni sul proprio nome e sul process id della root transaction;

- ***setInvoker*** *(sul web service del processo invocato)*: viene comunicato il nome del proprio processo al web service del processo invocato. Come mostrato in precedenza, questa informazione verrà letta dal nodo invocato nel punto di Before Receive;

- ***setActualPid*** *(sul web service del processo invocato)*: viene comunicato il process id della root transaction al web service del processo invocato. Come mostrato in precedenza, questa informazione verrà letta dal nodo invocato nel punto di Before Receive

- ***getVersion*** *(invocato sul proprio web service soltanto se l'epr restituito dal metodo map_XMLposition del punto precedente è nullo)*: il fatto che l'endpoint reference sia nullo all'interno della nostra implementazione significa che un componente invocherà un web service esterno per svolgere la logica della propria transazione. In questo caso, ai fini del criterio di Version Consistency, il processo BPEL e il suo web service esterno che esegue la logica di business (da non confondersi con il web service VersionConsistencyService) devono essere considerati come un unico componente. Anzi è proprio la modifica del web service esterno che determina l'aggiornamento del componente. La chiamata al metodo getVersion è fondamentale poiché attraverso questo metodo vengono svolti gli appositi controlli per determinare quale versione del web service è possibile utilizzare (in caso di concorrenza).

Se è possibile utilizzare una nuova versione siginifica che il componente viene aggiornato a runtime in maniera sicura, come previsto dalla nostra implementazione.

After invoke In caso di interceptor non relativo al processo che esegue la root transaction:

- ***getRootPid*** *(sul proprio web service)*: solita chiamata che avviene all'inizio del punto di intercettazione per ottenere il process id della root transaction.

In tutti gli interceptor, sia quelli relativi al processo che esegue la root transaction sia quelli relativi ai processi che eseguono le sottotransazioni:

- ***incrementPosition*** *(sul proprio web service)*: una volta eseguita l'attività di <Invoke>, bisogna aggiornare la posizione corrente sul file XML che contiene la sequenza delle chiamate. Quindi viene incrementata la posizione dello slot relativo alla transazione corrente.

Before reply Se il processo in esecuzione non svolge la root transaction:

- ***getRootPid*** *(sul proprio web service)*: come sempre all'inizio del punto di intercettazione ci si ricava il process id della root transaction;
- ***getInvokerEpr*** *(sul proprio web service)*: si ricava il nome del processo da cui si è stati invocati e tramite l'hashmap che rappresenta la configurazione statica (configurato come parametro dall'utente) si ottiene l'endpoint reference del suo web service;
- ***createOutEdge*** *(sul web service del processo invocante)*: viene chiamato il web service del processo invocante in modo che venga creato l'arco `Past` in uscita della configurazione dinamica; ciò indica che la transazione corrente è stata eseguita dal processo invocato;
- ***createInEdge*** *(sul proprio web service)*: viene creato l'arco `Past` corrispondente del punto precedente in ingresso alla propria configurazione dinamica;
- ***removeAutoEdges*** *(sul proprio web service)*: considerando che la transazione intercettata è al termine (attività di <Reply>), vengono rimossi gli autoanelli relativi alla transazione stessa. Praticamente vengono eliminati i due autoanelli con tempo `Future` e `Past` sia nella configurazione dinamica entrante sia in quella uscente;
- ***remove_future_Edges*** *(sul web service del processo invocante)*: si invoca il metodo remove_future_Edges sul web service del processo da cui si è stati invocati, per verificare se, in concomitanza con il termine della sottotransazione svolta, è possibile eliminare qualche arco `Future` dalla configurazione.

In caso di rimozione di eventuali archi, le chiamate al metodo procedono in modo ricorsivo sui vari web services dei nodi destinazione degli archi rimossi.

Per quanto riguarda questo punto di intercettazione (Before reply) va precisato che sarebbe stato più logico inserire alcune attività nel punto di intercettazione di After reply, ma ciò comportava degli evidenti problemi di sincronizzazione dovuti ad alcune limitazioni dell'engine di esecuzione dei processi e dell'architettura basata su web service: ad esempio nel caso in cui l'interceptor aspetta delle risposte da VersionConsistencyService esterni, le quali arrivano troppo in ritardo rispetto ai rapidi tempi di esecuzione dell'engine. In questo caso, per esempio si sarebbe dovuto introdurre altro traffico SOAP per sincronizzare web service e processi bpel, quindi ai fini della nostra implementazione è stato preferibile anticipare alcune operazioni come la rimozione degli autoanelli o la creazione dell'arco past nella fase di Before reply, causando un'imprecisione di pochi millisecondi.
After reply Attività svolte unicamente dallinteceptor del processo che esegue la root transaction:

- ***removeAutoEdges*** *(sul proprio web service)*: in questo punto la root transaction è terminata ed è stato restituito l'esito al client, quindi vengono rimossi gli autoanelli come avvenuto in precedenza nei web service dei processi che hanno eseguito le sottotransazioni.

Durante questo punto di intercettazione ricordiamo che l'interceptor del processo che svolge la transazione di root esegue anche la fase di Cleaning Up, presentata nel prossimo paragrafo.

4.3.4.3 Clean up

La fase di Clean Up avviene al termine di ogni transazione del sistema. Una volta che tutti i processi hanno terminato la loro sequenza di attività, la transazione distribuita è sicuramente completa e quindi questa fase serve per rimuovere gli archi rimanenti della configurazione dinamica.
Ovviamente si presuppone che siano presenti soltanto archi `Past` appartenenti alla transazione in questione poiché, essendo stati interamente eseguiti, tutti i processi avranno già invocato tutti i loro processi partner.
Questa fase può avvenire mentre altre transazioni del sistema sono ancora in corso, per cui la rimozione riguarda soltanto gli archi della configurazione dinamica con il parametro *transaction* che presenta l'id della root corrente. Ad esempio dopo la terminazione della transazione di root Portal.bpel con process id di valore 1, dovranno essere rimossi tutti gli archi restanti della configurazione dinamica con parametro *transaction* equivalente a Portal.bpel 1.

L'eliminazione degli archi viene avviata dall'interceptor del processo che esegue la root transaction nel punto di intercettazione After Reply. In questo determinato istante infatti la transazione generale è completa, ed è stato già restituito al client l'esito della transazione. L'unico processo attivo è la root transaction, che ha comunque terminato la sua sequenza di attività. Gli altri processi invece, avendo gestito delle sotto-transazioni, non sono sicuramente più istanziati.

Per avviare la fase di Clean Up, l'interceptor della root (quindi di Portal.bpel) invoca il metodo *remove_all_edges* del proprio web service. Questo metodo cerca tutti gli archi uscenti della configurazione dinamica (il web service della root transaction ovviamente non possiederà archi entranti poiché Portal.bpel non viene invocato da altri processi) con parametro *transaction* congruente con l'id della root transaction corrente.
Quando trova uno di questi archi lo rimuove, invoca il web service del nodo destinatario dell'arco in modo che possa rimuovere il corrispondente arco dinamico in ingresso (per essere rimosso in uscita da un nodo ovviamente deve essere rimosso anche in ingresso nell'altro) e avvia anche su tale nodo il metodo *remove_all_edges*. Si crea quindi un processo ricorsivo simile a quello utilizzato nella fase di inizializzazione, con i VersionConsistencyService dei componenti che si invocano l'un l'altro per rimuovere gli archi corretti, sfruttando sempre la propria conoscenza locale.
Con questo meccanismo quindi al termine di ogni transazione vengono eliminati tutti gli archi `Past` relativi ad essa.

4.3.5 Il meccanismo di sincronizzazione

Per un corretto funzionamento della nostra implementazione è fondamentale che qualsiasi processo, ad ogni punto di intercettazione del proprio Dynamo, sia in grado di conoscere il process id (PID) della root transaction. Questo PID viene infatti utilizzato da diversi metodi del VersionConsistencyService e in fase di chiamata è spesso previsto che l'interceptor lo inserisca direttamente come parametro. Di conseguenza nella struttura dati creata in ogni web service è presente un `ArrayList` di `String` chiamato rootPid_hash. Sfruttando questo array e la chiamata dall'interceptor ad un metodo presente anch'esso sul web service (getRootPid), è costantemente possibile ricavare l'id del processo di root relativo alla transazione distribuita in esecuzione.

Il process id della root transaction viene sempre propagato dal processo invocante al processo invocato prima di un'attività di <Invoke>; in questo modo durante la transazione distribuita qualsiasi processo che svolge una sottotransazione conoscerà il PID corretto grazie alla propria struttura dati.
Negli istanti precedenti l'attività di <Invoke>, il processo invocante comunica all'invocato anche il proprio nome (ad esempio semplicemente Portal.bpel). Infatti

durante i vari punti di intercettazione dell'interceptor è possibile sapere sempre chi si dovrà invocare grazie al file XML appositamente creato in fase di inizializzazione e all'array che memorizza la posizione corrente, ma al contrario non si conosce sempre da chi si è stati invocati. Questa informazione può essere utile per alcune chiamate tra web services durante lo svolgimento dell'algoritmo di Version Consistency, quindi viene memorizzata nella stessa stringa dell'array rootPid_hash che conterrà anche il process id della root transaction.

Ogni stringa dell'array rootPid_hash presente sui web service avrà la seguente struttura:
[PID root transaction - Nome Invoker - PID processo corrente] (a cui è associato il web service).
Grazie a questa struttura, se un processo invoca il metodo getRootPid del proprio web service passando come parametro il proprio PID (che ovviamente è sempre noto ad ogni componente) può ricavarsi in ogni istante di tempo le altre due informazioni.

Un altro aspetto interessante da analizzare è dovuto alla gestione della concorrenza. Bisogna infatti tenere conto del fatto che passano alcuni millesimi di secondo tra l'istante in cui le informazioni necessarie (PID root transaction e nome invoker) vengono inviate dal processo invocante al web service del processo invocato e l'istante in cui l'interceptor del processo invocato invoca il metodo del proprio web service per costruire la stringa e salvarla nell'apposito array. Se durante questo brevissimo intervallo di tempo un altro processo, appartenente ad un'altra transazione distribuita, cerca di passare le sue informazioni al web service ci sarebbero dei problemi perché alcune informazioni andrebbero perse. É stato quindi scelto di utilizzare una variabile di lock per impedire ad un processo di trasmettere le informazioni fino a quando non vengono memorizzate le altre già in transito nell'array rootPid_hash.

Ricapitolando, il meccanismo implementato è il seguente:

1. dopo aver verificato che la variabile di lock sul VersionConsistencyService del processo invocato sia settata a `false` (altrimenti bisogna attendere), durante la fase di Before Invoke il processo invocante setta a `true` il lock del web service del processo invocato in modo da prendere il controllo e poter passare correttamente le proprie informazioni;

2. sempre in fase di Before Invoke, il processo invocante comunica al web service del processo invocato il PID della root transaction; è importante notare che anche il processo invocante (se non è direttamente processo di root) ricava questa informazione grazie all'array rootPid_hash invocando a sua volta il proprio VersionConsistencyService;

3. in fase di Before Invoke, il processo invocante comunica al web service dell'invocato il proprio nome di processo (Invoker name);

4. il processo invocante invoca il processo invocato, che viene di conseguenza istanziato;

5. in fase di Before Start Process il processo invocato richiede al proprio web service le informazioni ricevute dall'invocante;

6. sempre in fase di Before Start Process il processo invocato chiama il metodo setRootPid del proprio web service e gli passa come parametri il proprio PID, il nome dell'invoker e il PID della root transaction (ottenuti in precedenza). In questo modo nell'array rootPid_hash del web service viene aggiunta la stringa composta dalla concatenazione di questi parametri. Da questo momento il processo invocato sarà in grado di ottenere sempre le informazioni invocando i metodi getRootPid o getInvokerName, a seconda di cosa si voglia ottenere dall'array;

7. il processo invocato, in fase di Before Start Process, una volta che la stringa è stata salvata correttamente nell'array rootPid_hash imposta a `false` il lock sul proprio web service in modo che esso possa ricevere nuove informazioni.

4.3.6 La rimozione degli archi future

Il metodo remove_future_edges è una parte fondamentale della nostra implementazione; la rimozione degli archi `Future` della configurazione dinamica durante la fase di Progress è determinante per il raggiungimento della Freeness dei vari componenti. Se un componente del sistema, relativamente ad una determinata transazione, non verrà più invocato da un altro componente collegato staticamente ad esso, l'arco `Future` che collega i due componenti deve essere rimosso.

Questo metodo di rimozione degli archi può essere invocato in tre istanti differenti durante l'esecuzione della transazione distribuita:

- in corrispondenza di un'invocazione tra processi, il processo invocato prima di iniziare il suo flusso di esecuzione nel punto di intercettazione Before Receve, invoca remove_future_edges sul web service del processo invocante;

- quando viene rimosso qualsiasi arco `Future` della configurazione dinamica durante la fase di Progress, viene invocato il metodo remove_future_edges sul web service del componente di destinazione dell'arco rimosso. In questo modo spesso viene eseguito ricorsivamente il metodo stesso, poiché la rimozione dell'arco `future`, se non si è già nella fase di Clean Up, è sicuramente stata effettuata proprio all'interno di remove_future_edges;

- se viene creato un arco `Past` nella configurazione dinamica, viene sempre invocato il metodo remove_future_edges sul web service del nodo sorgente dell'arco `Past` stesso.

Quando il metodo in questione viene avviato su un web service, effettua una serie di controlli su tutti gli archi della propria configurazione dinamica in uscita (relativa quindi al nodo associato al web service stesso). Per ogni arco di tipologia `Future` appartenente alla transazione distribuita corrente, si verifica se è possibile effettuare la rimozione eseguendo i seguenti controlli:

- si verifica che non ci siano archi entranti nella propria configurazione dinamica con tempo `Future` relativi alla stessa transazione. Se infatti esiste almeno un arco di questo tipo, significa che il componente verrà utilizzato di nuovo all'interno della transazione distribuita, e quindi a sua volta avrà la necessità di invocare il processo destinatario dell'arco `Future` di cui si sta valutando la rimozione;

- si effettua una verifica attraverso il parsing del file XML del nodo relativo al web service che sta eseguendo remove_future_edges. In particolare viene controllato che il nodo destinazione dell'arco `Future` da rimuovere non sia nell'elenco dei nodi che verranno invocati dalla posizione corrente in poi del file XML. Questo controllo è fondamentale per verificare se in futuro un nodo potrà essere invocato di nuovo dal nodo sorgente o meno.

Quando viene eliminato un arco `Future` della configurazione dinamica viene immediatamente invocato il metodo remove_future_edges sul web service del nodo destinazione dell'arco rimosso. Quindi si crea una sequenza di chiamate ricorsive in modo da verificare istantaneamente se la rimozione di un arco può causare la rimozione di altri archi sui vari web services; con questo metodo la freeness dei processi viene raggiunta il prima possibile.

Capitolo 5

Validazione del progetto

In questo capitolo daremo delle misure di validazione del nostro progetto: analizzeremo due casi d'uso per mettere in evidenza le caratteristiche salienti della nostra implementazione e infine mostreremo delle statistiche che misurano le prestazioni e che ci hanno permesso di capire dove poter intervenire per possibili miglioramenti.

5.1 Esecuzione di transazioni consecutive

Come primo caso d'uso consideriamo il caso in cui il client invoca il processo Portal.bpel sequenzialmente, cioè effettua la seconda chiamata aspettando che la prima transazione distribuita sia terminata, ovvero dopo che Portal ha restituito il risultato.
Mostriamo passo-passo gli archi dinamici creati e rimossi dai VersionConsistencyService, analizzando anche cosa succede nei vari punti di intercettamento dei componenti del nostro sistema.

Portal - *before start process*
Il client invia la richiesta verso il nodo Portal: l'engine riceve il messaggio SOAP in ingresso e istanzia il processo che verrà eseguito, ma l'interceptor blocca l'inizio della transazione poichè in questa fase viene effettuata la fase di set up dell'algoritmo.
La fase di inizializzazione viene dunque attuata a processi non ancora in esecuzione: l'interceptor di Portal passa il controllo al VersionConsistencyService che, attraverso la lettura dei propri file *.bpel* e *.pdd* (configurazione statica locale), costruisce i primi due archi dinamici in uscita, Portal $\xrightarrow[portal.bpel1]{future}$ Auth e Portal $\xrightarrow[portal.bpel1]{future}$ Proc.
Successivamente il VersionConsistencyService interagisce con i due nodi staticamente dipendenti, Auth e Proc, i quali creano gli stessi archi sulla loro configura-

zione dinamica, stavolta però salvati come archi in ingresso.
Fatto ciò, ricorsivamente Auth e Proc effettuano le stesse operazioni di Portal, andando eventualmente a creare i loro archi dinamici in uscita; nel nostro caso verrà creato solo Proc $\xrightarrow[portal.bpel1]{future}$ Auth , poichè Auth staticamente non richiede servizio ad altri nodi del sistema (fa richiesta solo al proprio web service che implementa la logica di business, ma come già detto, non essendo un vero e proprio nodo, non è interessato dall'algoritmo di riconfigurazione).
La fase di set up termina e il controllo ripassa all'interceptor di Portal; la configurazione del sistema è rappresentata in figura 5.1.

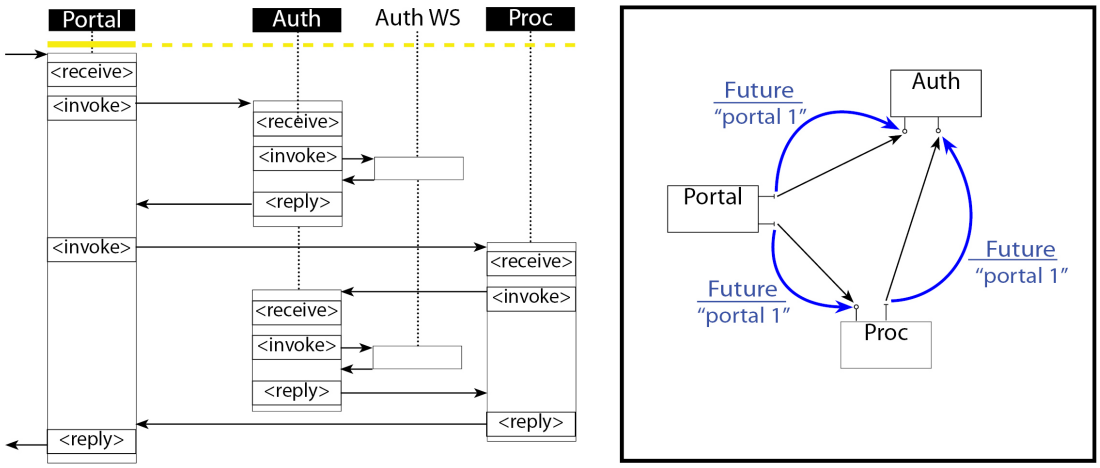


Figura 5.1: Situazione archi: before start process di Portal.

Portal - *before receive/before invoke*

Dopo la fase di *before start process*, il processo Portal può cominciare la propria esecuzione; prima dell'attività di <receive> vengono però creati gli autoanelli Portal $\xrightarrow[portal.bpel1]{future}$ Portal e Portal $\xrightarrow[portal.bpel1]{past}$ Portal, sia in ingresso che in uscita, per un totale di 4 archi dinamici.
Dopo la <receive>, il processo continua con l'attività di <invoke>: nel punto di *before invoke* vengono effettuate delle operazioni necessarie all'interceptor per gestire archi, concorrenza e aggiornamento che abbiamo già visto nel capitolo 4 (e che analizzeremo nei paragrafi successivi), mentre per quanto riguarda la configurazione dinamica non cambia niente. La configurazione appena prima dell'activity di <invoke> è mostrata in figura 5.2.
Dopo questa activity Portal invoca Auth passandogli il controllo dell'esecuzione

dell'intera transazione distribuita.

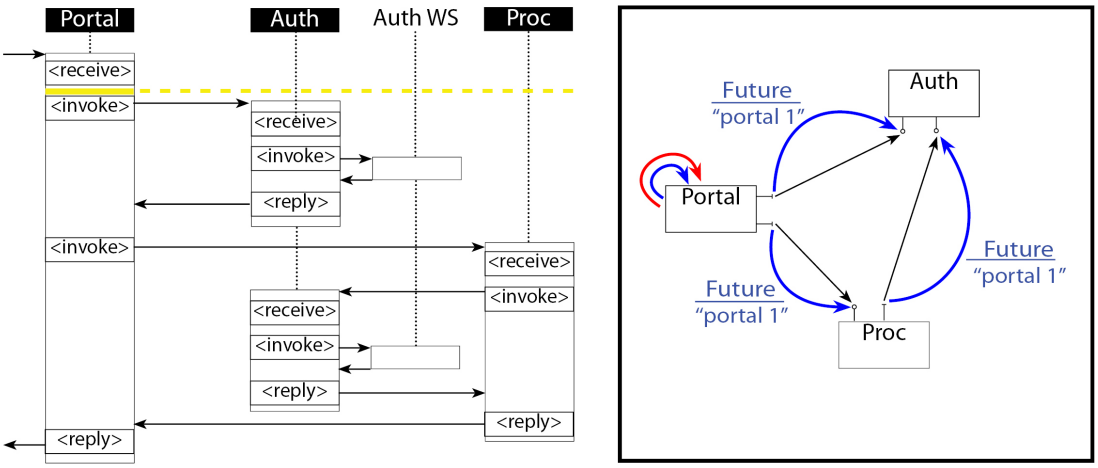


Figura 5.2: Situazione archi: before invoke di Portal.

Auth - *before receive/before invoke*

Nel punto di *before start process* di Auth si effettuano sostanzialmente delle operazioni legate alla gestione della concorrenza, mentre, come su Portal è prima della <receive> che vengono creati i quattro autoanelli (due ingresso e due in uscita): Auth $\xrightarrow[portal.bpel1]{future}$ Auth e Auth $\xrightarrow[portal.bpel1]{past}$ Auth.

In più, siccome Auth è stato chiamato da Portal, quest'ultimo eseguirà il metodo $remove\_future\_edges$ sul proprio VersionConsistencyService per verificare nella sua configurazione dinamica in uscita se ci sono archi future eliminabili. In questo caso sono presenti Portal $\xrightarrow[portal.bpel1]{future}$ Auth e Portal $\xrightarrow[portal.bpel1]{future}$ Proc: tra questi è rimosso l'arco future verso Auth; questo succede perchè Portal, leggendo il proprio file XML (che ha il compito di tracciare la sequenza di invocazioni future), è sicuro che non chiamerà più Auth. Il metodo $remove\_future\_edges$ viene poi eseguito ricorsivamente anche su Auth ma in questo caso non avviene alcuna rimozione poichè non ci sono archi future nella sua configurazione dinamica in uscita.

Procedendo con l'esecuzione di Auth, effettuata l'<invoke> verso il web service Auth_WS, la situazione degli archi dinamici è simile a quella rappresentata in figura 5.2, con l'aggiunta degli autoanelli su Auth e senza l'arco Portal $\xrightarrow[portal.bpel1]{future}$ Auth.

Auth - *before/after reply*

In concomitanza con l'activity di <reply>, siccome Auth sta per terminare la propria esecuzione, viene creato l'arco dinamico Portal $\xrightarrow[portal.bpel1]{past}$ Auth (ovviamente anche in questo caso, sia nella configurazione in uscita di Auth che nella configurazione in ingresso di Portal).
In più, vengono rimossi gli autoanelli da Auth e viene eseguito nuovamente il metodo $remove\_future\_edges$ su Portal, ma in questo caso l'unico da controllare è Proc $\xrightarrow[portal.bpel1]{future}$ Auth, che, come già notato al passo precedente, non viene rimosso. La configurazione dinamica è mostrata in figura 5.3.

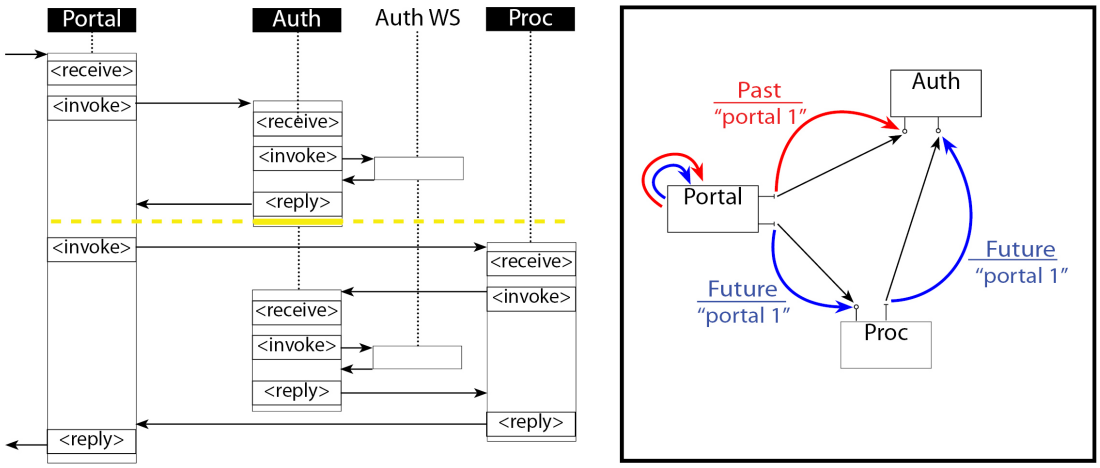


Figura 5.3: Situazione archi: after reply di Auth.

Proc - *before receive/before invoke*

A questo punto il controllo ritorna a Portal che chiama Proc tramite la seconda <invoke>.
Le operazioni effettuate dall'algoritmo sono le medesime rispetto al caso precedente, per cui Proc comincia la propria esecuzione e in concomitanza della <receive> crea gli autoanelli Proc $\xrightarrow[portal.bpel1]{future}$ Proc e Proc $\xrightarrow[portal.bpel1]{past}$ Proc; infine i VersionConsistencyService di Portal e Proc si chiamano a vicenda tramite il metodo $remove\_future\_edges$: in questo caso l'arco Portal $\xrightarrow[portal.bpel1]{future}$ Proc viene rimosso poichè Portal non richiamerà in futuro Auth, diversamente da quanto accade per l'arco Proc $\xrightarrow[portal.bpel1]{future}$ Auth, che non viene rimosso poichè ovviamente

Auth non è ancora stato chiamato da Proc.

Auth - *before/after reply*

Auth viene quindi chiamato da Proc e istanziato una seconda volta: si creano gli autoanelli su di esso e tramite il metodo $remove\_future\_edges$ viene rimosso l'arco Proc $\xrightarrow[portal.bpel1]{future}$ Auth.
Auth effettua la chiamata al proprio web service e, ricevuta la risposta, si appresta ad eseguire la <reply> verso Proc.
A questo punto, come successo prima, vengono rimossi gli autoanelli e creato l'arco dinamico Proc $\xrightarrow[portal.bpel1]{past}$ Auth.
A differenza della prima istanza, Auth questa volta raggiunge lo stato di *"free"*, poichè presenta solo archi past in ingresso rispetto alla transazione root *portal.bpel 1*, come si nota in figura 5.4.
Il nodo Auth è quindi pronto (safe) per essere aggiornato, ma discuteremo in seguito come viene gestito l'update.

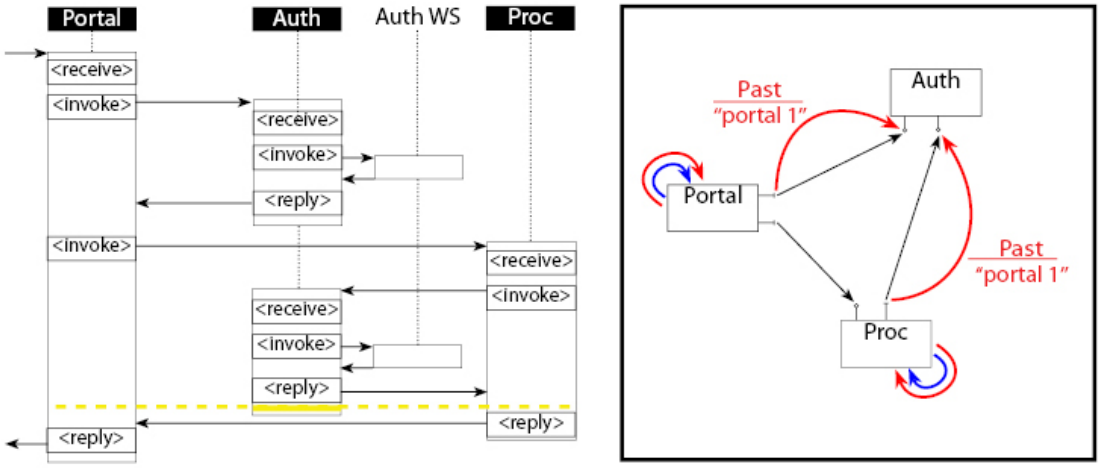


Figura 5.4: Situazione archi: after reply di Auth, seconda istanza.

Proc - *before/after reply*

Dopo la <reply> di Auth, è Proc che deve eseguire la propria <reply> verso Portal.
Vengono quindi rimossi gli autoanelli e aggiunto l'arco Portal $\xrightarrow[portal.bpel1]{past}$ Proc: anche Proc è *free* rispetto alla transazione root.

Portal - *before/after reply*

Dopo la <reply> di Proc, è Portal che deve eseguire la propria <reply> verso il client. Per prima cosa sono rimossi gli autoanelli e successivamente, siccome Portal è il nodo che esegue la transazione root, viene chiamato il metodo $remove\_all\_edges$, che ha il compito di effettuare la fase di cleaning up dell'algoritmo. In questa fase sono rimossi gli archi past che sono rimasti nella configurazione dinamica: Portal $\xrightarrow[portal.bpel1]{past}$ Auth e Portal $\xrightarrow[portal.bpel1]{past}$ Proc; il metodo ricorsivamente chiama $remove\_all\_edges$ sui nodi destinazione, quindi è rimosso anche l'arco Proc $\xrightarrow[portal.bpel1]{past}$ Auth. Il processo termina e il sistema ritorna alla configurazione statica iniziale, senza archi dinamici (figura 5.5).

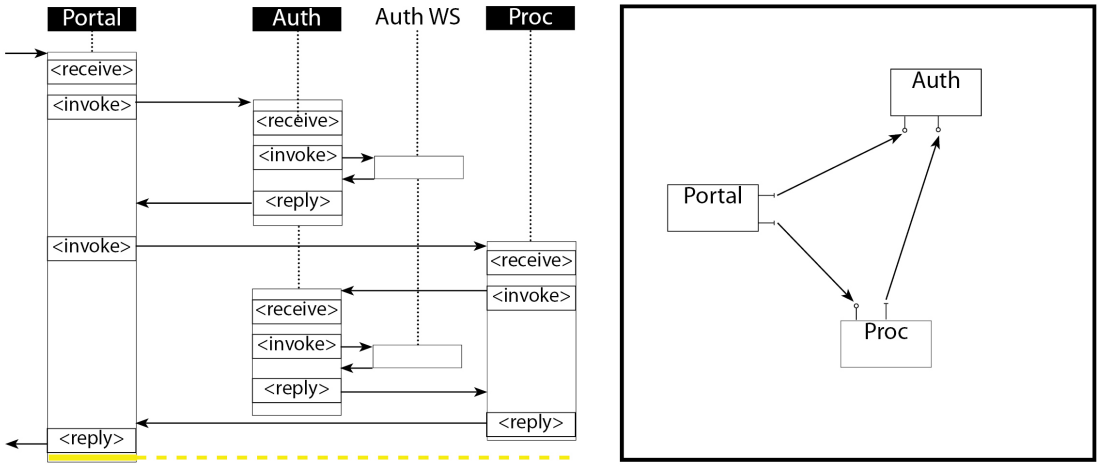


Figura 5.5: Situazione archi: after reply di Portal.

La seconda chiamata del client darà luogo ad una seconda transazione root denominata *Portal.bpel 2* e gli archi costruiti saranno esattamente gli stessi della transazione 1.

5.2 Esecuzione di transazioni concorrenti

Se il client non aspetta la fine della root transaction per effettuare chiamate successive alla prima, il nostro sistema è in grado di gestire correttamente l'algoritmo creando archi dinamici coesistenti sui nodi identificati da diverse root transactions. Nel seguito mostreremo cosa succede nel caso di due transazioni concorrenti, andando a mostrare la configurazione dinamica in due punti temporali che ci permetteranno di discutere circa il nostro approccio alla concorrenza

e la strategia di aggiornamento adottata.

5.2.1 Gestione della concorrenza

La gestione della concorrenza non è stato un problema di facile risoluzione: come è facile intuire, per costruire gli archi è necessario identificare la root transaction durante l'esecuzione di una qualsiasi istanza di un processo: noi abbiamo deciso di utilizzare il pid della root transaction.
Se intercettiamo l'esecuzione parallela con due istanze dello stesso processo, l'engine è in grado di capire quale sia il pid (tra i due in esecuzione) rispetto all'activity che stiamo intercettando; quindi è semplice capire il valore della root transaction nel caso di Portal; ma non è invece così immediato capirlo nel caso di una sotto transazione, ad esempio Proc (figura 5.6).
Il problema è stato quindi la propagazione del valore del root pid su tutti i nodi del sistema distribuito, cercando di mantenere la località dell'algoritmo: ogni istanza di un processo deve conoscere, a runtime, il pid del processo root che ha generato l'intera transazione distribuita.

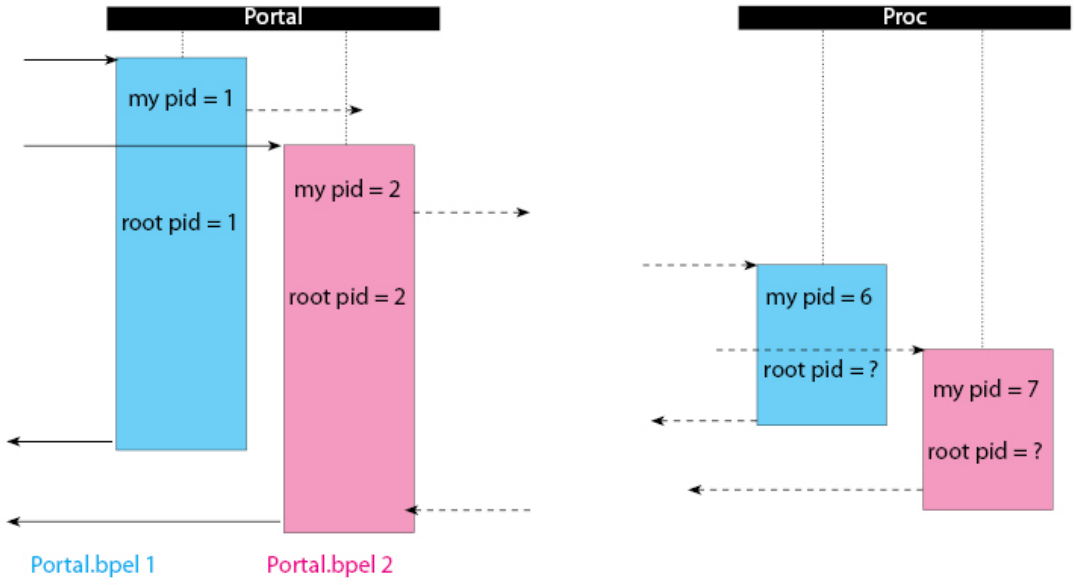


Figura 5.6: Il problema della propagazione del root pid.

La nostra soluzione, ampiamente mostrata nel capitolo 4 nella sezione che descrive il rootPid_hash, ci permette di capire cosa succede nel nostro caso d'uso. Nella fase di inizializzazione Portal dà inizio alla catena di archi future associando il proprio nome, la root transaction *portal.bpel*, al proprio process id, in questo

caso quindi diventa *portal.bpel 1*; il pid della root transaction viene poi propagato su ogni arco associato alla prima istanza dell'intera transazione distribuita.
All'altezza del punto di *before start process* della prima istanza abbiamo quindi una situazione che ricalca quella vista per il caso d'uso che considera solo transazioni consecutive: la configurazione statica, durante l'esecuzione si arricchirà quindi degli stessi archi visti prima.

Consideriamo ancora la prima transazione distribuita *portal.bpel 1*, nel punto di *before invoke* di Portal: l'interceptor legge il pid del processo in esecuzione che è pari ad 1; questo valore dovrà essere inviato all'interceptor di Auth (figura 5.7).
Per fare questo Portal si mette in attesa del rilascio del lock sulla risorsa condivisa

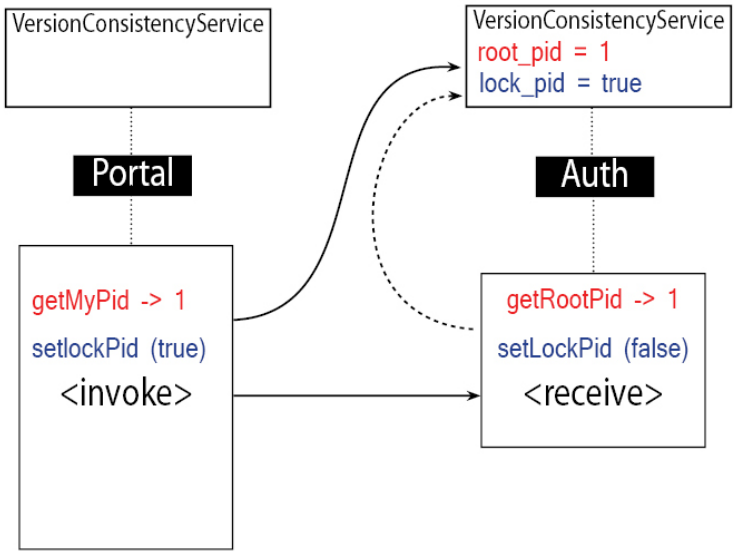


Figura 5.7: Sintesi del passaggio del process id della transazione root, da Portal ad Auth.

del VersionConsistencyService di Auth che ha il compito di salvare temporaneamente il valore del root pid: in questo caso il lock è libero poichè il sistema si appresta ad eseguire la prima istanza di Auth e dunque non possono esistere altri componenti che stanno effettuando la stessa operazione. Così Portal scrive il valore 1 su Auth; inoltre scrive anche il proprio nome (Portal.bpel) in qualità di invoker.
Auth, una volta istanziato, nel punto di *before receive* legge dal proprio VersionConsistencyService queste due informazioni, ovvero `1` e `Portal.bpel`, e successivamente chiama il metodo *setRootPid* passandole come parametri. Nel web service sono così salvate queste due stringhe nella prima cella dell'ArrayList rootPid_hash come stringa concatenata: `1-Portal.bpel-1`. Fatto questo, il Ver-

sionConsistencyService di Auth rilascia il lock per permettere la propagazione dei root pid di eventuali transazioni distribuite successive.
Grazie a questo meccanismo, l'interceptor in seguito leggerà dal proprio VersionConsistencyService il valore 1 (identificato dal proprio pid 1 e dal nome dell'invoker *Portal.bpel* nel rootPid_hash) e lo utilizzerà per etichettare tutti gli archi creati in questa fase, ovvero gli autoanelli, e per verificare se può rimuovere l'arco Portal $\xrightarrow[portal.bpel1]{future}$ Auth creato in precedenza.

Ad un certo punto, come si può notare in figura 5.8, il client chiama una seconda volta Portal, per cui l'interceptor andrà a leggere nuovamente il process id (che stavolta sarà pari a 2) e darà luogo ad una nuova fase di inizializzazione per la seconda transazione distribuita, con gli archi etichettati da *portal.bpel 2*.

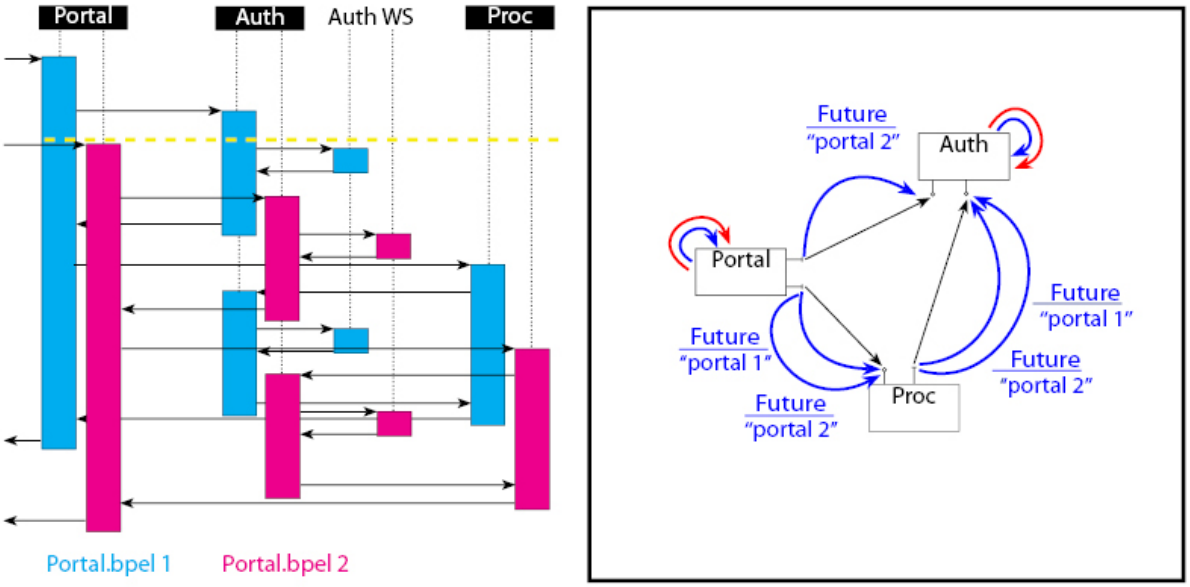


Figura 5.8: Situazione archi: Portal esegue la seconda root transaction.

Portal a questo punto esegue la seconda transazione concorrentemente alla prima; l'engine Active-Bpel è in grado di fare questo senza problemi e l'interceptor è quindi capace di reperire informazioni dinamicamente anche su questa seconda root transaction, in particolare il pid dell'istanza del processo intercettato.
Ad esempio, quando *portal.bpel 2* arriva alla propria <invoke>, andrà a leggere il proprio pid che stavolta sarà pari a 2.
Anche questa volta Portal si mette in attesa del rilascio del lock su Auth e appena possibile scriverà il valore 2 sul suo VersionConsistencyService; Auth a sua volta, in concomitanza della <receive> leggerà il valore 2 e creerà gli autoanelli Auth $\xrightarrow[portal.bpel2]{future}$ Auth e Auth $\xrightarrow[portal.bpel2]{past}$ Auth.

In questo esatto istante temporale, come si può notare analizzando la figura 5.8, abbiamo due istanze concorrenti sia su Portal che su Auth: in entrambe le configurazioni dinamiche avremo 8 autoanelli (non rappresentati in figura): 4 autoanelli (2 archi in ingresso e due in uscita) per *portal.bpel 1* e 4 autoanelli per *portal.bpel 2*.
La configurazione dinamica del sistema sarà quindi formata da archi identificati da due root transaction diverse, che evolveranno come nel caso di transazioni singole ma che incideranno contemporaneamente ogni qual volta si dovrà verificare la freeness per l'aggiornamento di un componente.

Gestire la concorrenza attuando il passaggio del pid in questo modo presenta pro e contro, ed è stato uno dei problemi di maggior rilevanza durante l'implementazione.
Lo svantaggio principale deriva dal fatto che è stato necessario introdurre dei lock sulle risorse condivise tra più processi: il lock previene sovrascritture tra l'activity di <invoke> del chiamante e la <receive> del chiamato quando si scambiano il valore del root pid. Il problema è che implementare questa struttura bloccante ha aumentato il numero di messaggi scambiati tra i componenti del sistema, e questo ha introdotto dei ritardi; inoltre se il nodo chiamato dovesse danneggiarsi/bloccarsi prima di rilasciare il lock, tutti gli altri nodi resterebbero in attesa infinita. D'altro canto il tempo in cui la risorsa condivisa resta bloccata è decisamente ridotto, poichè è il tempo che intercorre tra una <invoke> e una <receive> (e dipende sostanzialmente dalla rete su cui si appoggia il sistema distribuito).
Il vantaggio è che invece la nostra soluzione è semplice non richiedendo strutture dati particolarmente complesse e quindi non aggiunge overhead in termini di calcolo locale; inoltre è facilmente estendibile per sviluppi futuri, ad esempio per aggiungere sicurezza e mantenere attivo l'algoritmo in caso uno dei nodi vada in crash.
Una possibile alternativa sarebbe quella di inoltrare il root pid arricchendo il *SOAP Header* dei messaggi scambiati tra i processi BPEL, che non implicherebbe alcun messaggio aggiuntivo e neanche un lock visto che non sarebbe più necessario appoggiarsi al VersionConsistencyService condiviso. Il problema è che l'interceptor non sempre riesce ad intercettare i messaggi SOAP scambiati, e sarebbe necessario ad esempio, fare del parsing sui messaggi in ingresso a Dynamo; oppure aggiungere un *SOAP handler*, ad esempio per gestire delle code per non perdere messaggi nell'intento di intercettare tutti i messaggi nei punti desiderati; questo creerebbe comunque dei pesanti ritardi sull'esecuzione delle transazioni BPEL, per cui dovremmo fare un'analisi dettagliata per valutare gli effettivi benefici di questa soluzione.

5.2.2 Gestione dell'aggiornamento

Aggiornare un componente di un sistema distribuito significa sostituirlo con una nuova versione: nel nostro caso i componenti sono i processi BPEL e l'aggiornamento di un nodo dovrebbe essere fatto sostituendo il processo con un altro processo che mantenga inalterati wsdl, partnerlinks, variabili, e tutto ciò che è necessario agli altri processi BPEL per poter interagire correttamente con esso.
Ma per fare questo in modo pulito e soprattutto a runtime, aggiornare un processo BPEL significa ridursi ad aggiornare la sua logica di business, ovvero il web service che implementa il servizio vero e proprio.
Nel nostro esempio il nodo che implementa la logica di business è Auth, e il web service interessato dall'aggiornamento è *Auth_service.jws (indicato anche con Auth_WS)*: ma come implementare un aggiornamento del web service a runtime e soprattutto in modo automatico?
Siccome il processo Auth, prima di effettuare l'invoke, legge l'indirizzo di *Auth_service* all'interno del partner link associato ad esso, la nostra idea è stata quella di modificare l'endpoint reference del partner link a runtime. Questo compito è relegato all'interceptor, che nel punto di intercettamento *before invoke*, dinamicamente andrà a modificare l'indirizzo del servizio esterno, facendo puntare il partner link verso una nuova versione del servizio, come si vede in figura 5.9.

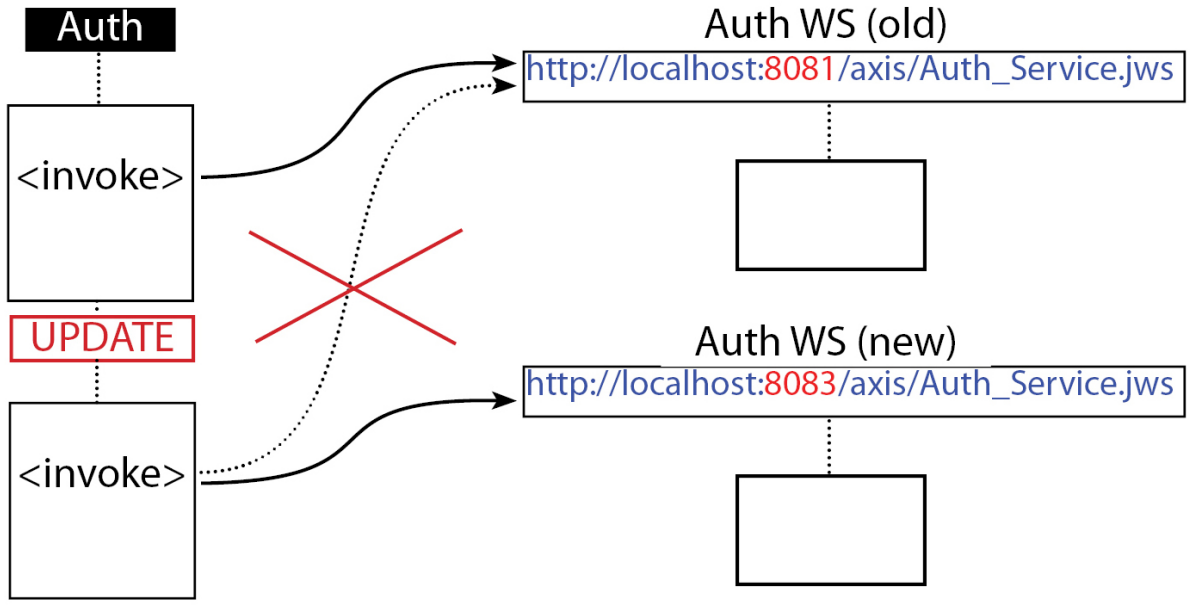


Figura 5.9: Aggiornamento del processo Auth.

Come visto nel capitolo 3, per poter aggiornare un componente del sistema distribuito, esso deve prima raggiungere lo stato di *free*: un componente è free

rispetto ad una transazione root T se non esiste una coppia di archi future e past, entranti nel nodo considerato ed etichettati da T.
Il vantaggio di questo approccio è che l'update dell'indirizzo endpoint è in realtà fatto appena il nodo raggiunge lo stato di free: il nostro codice prevede un controllo locale della freeness ogni volta che rimuoviamo un arco in ingresso nella configurazione dinamica; se in concomitanza della rimozione di un arco il nodo diventa free, subito viene settato un flag che permetterà a qualsiasi interceptor invocante di puntare alla nuova versione.
Grazie a questo sistema possiamo aggiornare un nodo in modo rapido e senza aggiungere particolari funzionalità che avrebbero appesantito la gestione della riconfigurazione.

In particolare, questo tipo di implementazione ci ha permesso di applicare la strategia più funzionale nel raggiungimento della freeness, la strategia di *Concurrent Versions*.
Ma andiamo con ordine, presentando pregi e difetti delle tre strategie per il raggiungimento della freeness, *Waiting for freeness*, *Concurrent Versions* e *Blocking for freeness*, motivando la scelta fatta in sede di implementazione.
Per confrontare le tre possibilità, prendiamo come esempio la figura 5.10: l'istante di intercettamento è in concomitanza della activity di <reply> da parte della seconda istanza di Auth, nella prima transazione distribuita.
Auth raggiunge la freeness (rispetto alla transazione root *portal.bpel 1*) in due

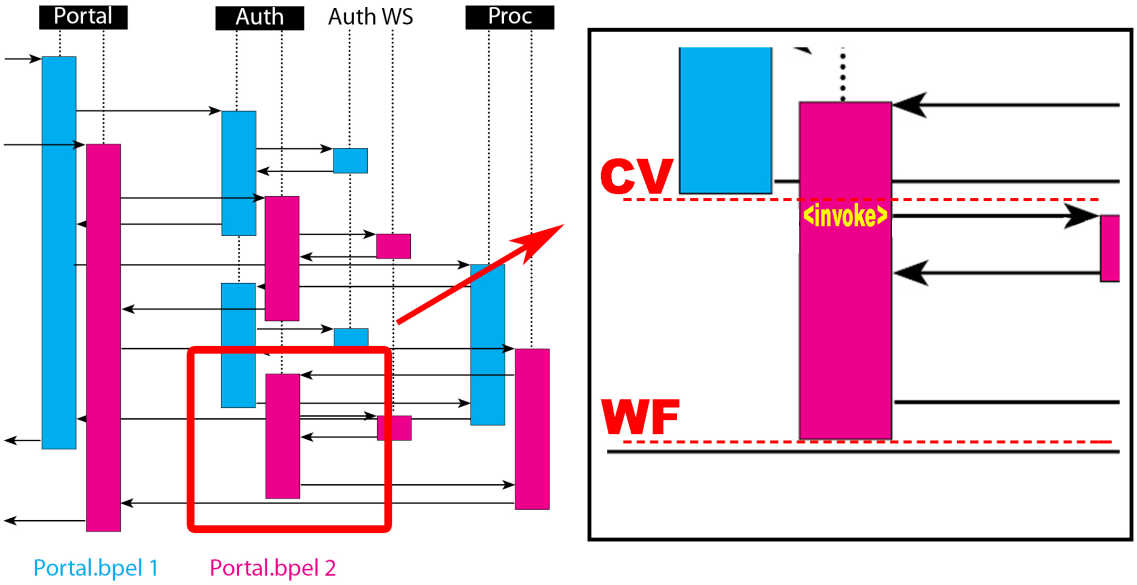


Figura 5.10: Freeness del nodo Auth utilizzando differenti criteri per il raggiungimento della freeness.

momenti diversi, a seconda se si usa la versione *waiting for freeness* o *concurrent*

versions, come si può vedere in figura.
Nel caso della strategia di *waiting for freeness (WF)*, Auth raggiunge lo stato di free rispetto a *portal.bpel 1* nello stesso istante (CV) della strategia *concurrent versions*; ma Auth non è free rispetto a *portal.bpel 2*, infatti sta eseguendo una transazione che comporta la presenza dei due autoanelli Auth $\xrightarrow[portal.bpel2]{future}$ Auth e Auth $\xrightarrow[portal.bpel2]{past}$ Auth, che compongono una coppia di archi future/past in ingresso identificati dalla stessa root transaction. Del resto aggiornare in questo istante sarebbe un errore poichè, come si può notare in figura, l'invoke punterebbe verso la versione aggiornata del servizio esterno, e questo non garantirebbe la consistenza del sistema, poichè precedentemente aveva invocato la versione vecchia. Se dunque applichiamo il criterio di *waiting for freeness*, dovremo aspettare che il componente sia free rispetto a tutte le transazioni in corso sul nodo, fino all'istante (WF) di figura 5.10.
Se invece utilizziamo la strategia di *concurrent versions*, l'aggiornamento di *Auth* è possibile anche senza aspettare la terminazione della seconda transazione: l'endpoint viene mantenuto per questa transazione, e l'esecuzione continua sulla vecchia versione, mentre tutte le nuove transazioni distribuite possono già utilizzare la nuova: ogni invoke successiva punterà al nuovo endpoint reference.
La versione di *Blocking for freeness* (BF), in questo caso, bloccherebbe l'esecuzione della prima transazione distribuita, aspettando la fine della seconda per aggiornare l'endpoint comune a tutte le transazioni eseguite da Auth; per questo l'aggiornamento sarebbe ritardato fino all'istante (WF), cioè l'istante in cui Auth è free anche rispetto a *Portal.bpel 2*, e quindi non sarebbe vantaggioso in termini di latenze, anzi sarebbe peggio anche rispetto alla strategia di *Waiting for freeness* poichè la prima transazione distribuita rimarrebbe bloccata per un certo lasso di tempo.

Riprendendo il nostro caso d'uso, analizziamo cosa succede esattamente al momento indicato in figura 5.11.
Nel punto di intercettamento, Auth ha terminato la transazione relativa alla root *portal.bpel 1*, mentre sta ancora eseguendo la transazione parallela relativa a *portal.bpel 2*.
Nel punto di reply si eliminano gli autoanelli Auth $\xrightarrow[portal.bpel1]{future}$ Auth e Auth $\xrightarrow[portal.bpel1]{past}$ Auth poichè Auth ha terminato la propria esecuzione rispetto a *portal.bpel 1* e viene lanciato il metodo $remove\_future\_edges$ per verificare ed eventualmente eliminare alcuni archi future, sempre etichettati da *portal.bpel 1*.
Come detto, il nostro codice prevede il controllo locale della freeness sul VersionConsistencyService, in concomitanza di ogni arco rimosso. Dopo la cancellazione dell'autoanello Auth $\xrightarrow[portal.bpel1]{future}$ Auth la configurazione dinamica in ingresso ad Auth è composta da:

- Portal $\xrightarrow[portal.bpel1]{past}$ Auth, Proc $\xrightarrow[portal.bpel1]{past}$ Auth; Auth è free rispetto alla transazione *portal.bpel 1*.
- Portal $\xrightarrow[portal.bpel2]{past}$ Auth, Auth $\xrightarrow[portal.bpel2]{past}$ Auth, Auth $\xrightarrow[portal.bpel2]{past}$ Auth non è free rispetto alla transazione *portal.bpel 2*.

Il controllo della freeness è eseguito sia per gli archi della prima transazione che per gli archi della seconda e siccome Auth risulta free rispetto alla *portal.bpel 1*, si effettuano le seguenti operazioni: viene sostituito l'endpoint della stringa *currentEpr* che conterrà ora l'endpoint della nuova versione (`http://localhost:8083/axis/Auth_Service.jws`), mentre il pid `1` della prima transazione distribuita è cancellato dall'ArrayList *transactionsEprOld* e spostato nell'ArralyList *transactionsEprNew*. In questo modo la transazione *portal.bpel 2* continuerà ad utilizzare la vecchia versione mentre tutte le transazioni distribuite successive utilizzeranno la nuova.
Infine viene creato l'arco past Proc $\xrightarrow[portal.bpel1]{past}$ Auth; la configurazione dinamica è mostrata in figura 5.11.

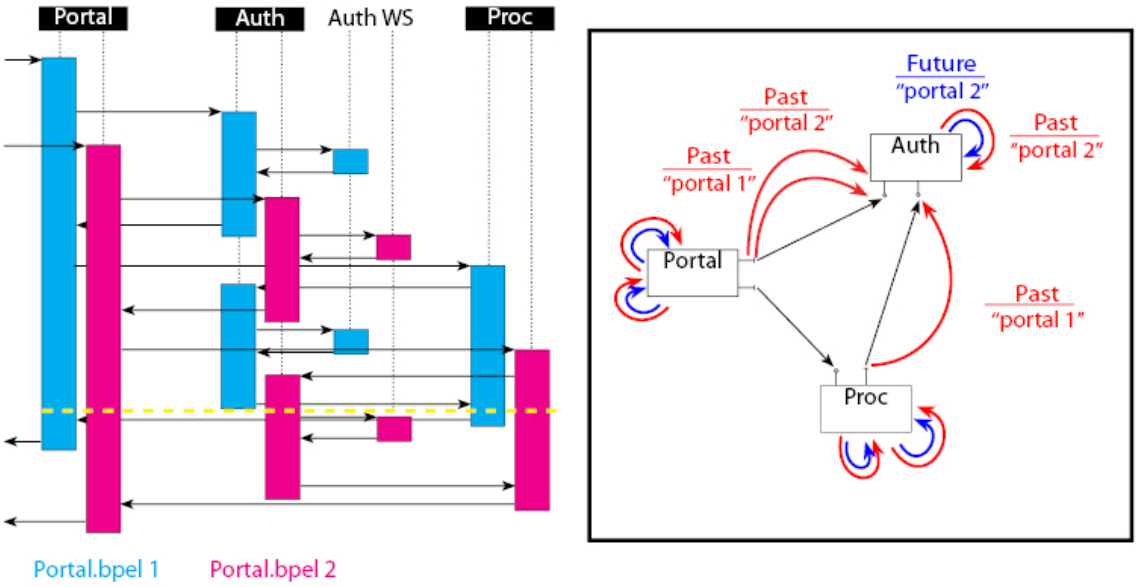


Figura 5.11: Auth è free rispetto a *Portal.bpel 1*.

A questo punto, l'unica transazione in corso su Auth continua la propria esecuzione e, ad un certo punto, incontra l'activity di <invoke>: siccome il pid 2 non è stato spostato sull'ArrayList *transactionsEprNew*, *portal.bpel 2* usa ancora la vecchia versione. In questo modo l'interceptor non modifica l'endpoint del partner link, che è settato sulla vecchia versione di *Auth_ WS*; questo permetterà

ad Auth di continuare ad utilizzare la vecchia versione per preservare la consistenza del sistema.
Invece, se arrivasse una chiamata concorrente legata ad una ipotetica nuova root transaction *portal.bpel 3*, l'endpoint del partner link di *Auth_ WS* punterà già alla nuova versione: avremmo quindi in esecuzione due versioni concorrenti dello stesso servizio *Auth_ WS*.
Quando termina anche la seconda transazione su Auth, ogni nuova chiamata andrà sulla nuova versione, per cui sarà possibile rimuovere manualmente e definitivamente la vecchia versione, mantenendo solamente il deploy della nuova.

5.3 Statistiche sul numero di messaggi

Una misura indicativa dell'efficienza per i sistemi distribuiti è il numero di messaggi scambiati tra i vari componenti: pochi messaggi corrispondono a un minore overhead sulla rete e quindi ad una maggiore efficienza.
Nel nostro esempio, il sistema distribuito è costituito da più processi BPEL, ciascuno eseguito da un engine Active-Bpel, per cui la misura di efficienza sarà effettuata contando i messaggi SOAP scambiati tra i vari engines, in particolare analizzeremo:

- numero di messaggi SOAP scambiati tra gli Active-Bpel engines senza le funzionalità dell'algoritmo;
- numero di messaggi SOAP scambiati tra gli Active-Bpel engines con l'aggiunta delle funzionalità dell'algoritmo.

Ovviamente l'efficienza è misurata sull'esecuzione di un'intera transazione distribuita su tutti i nodi del nostro sistema. Nel primo caso si misurano sostanzialmente i messaggi scambiati all'interno del processo BPEL del nostro esempio; nel secondo caso si contano i messaggi scambiati CON l'algoritmo in funzione. Questo perchè gli Active-Bpel engines sono arricchiti dalle funzionalità di Dynamo e del VersionConsistency web service, che per interagire devono per forza di cose scambiarsi dei messaggi SOAP; per questo motivo faremo un'analisi più dettagliata dei messaggi scambiati tra i componenti interni dei nodi del nostro CBDS, in particolare:

1. numero di messaggi SOAP scambiati tra un Dynamo (SOAP request) e il suo VersionConsistencyService (SOAP response);
2. numero di messaggi SOAP scambiati tra un Dynamo (SOAP request) e i VersionConsistencyService con cui interagisce (SOAP response);
3. numero di messaggi SOAP scambiati tra un VersionConsistencyService (SOAP request) e gli altri VersionConsistencyService (SOAP response).

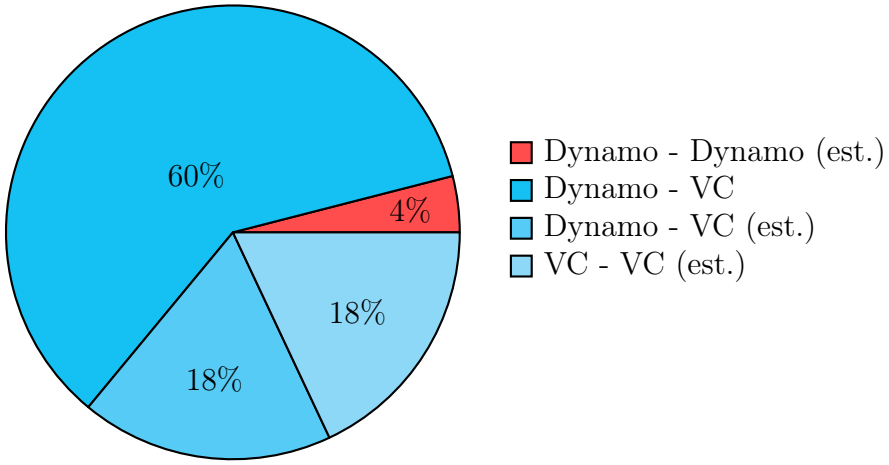
Nel primo caso si misurano i messaggi che ogni interceptor scambia con il proprio VersionConsistencyService, nei vari punti di intercettazione per la gestione degli archi dinamici. Come visto nel cap 4, l'interceptor in concomitanza di ogni activity chiama il proprio VersionConsistencyService che risponde in modo sincrono, generando del traffico di rete che possiamo interpretare come traffico "locale".
Una misura del traffico non locale è data dagli ultimi due casi: per esempio, nella creazione di nuove sottotransazioni, l'interceptor del nodo chiamante deve passare i parametri che identificano il nome dell'invoker e il pid della transazione root al VersionConsistencyService del nodo invocato. Oppure il VersionConsistencyService del nodo invocato deve chiamare quello dell'invocante per verificare la possibilità di rimozione degli archi future attraverso la chiamata ricorsiva sul metodo remove_future_edges.
Tutto questo traffico di rete è riassunto nelle statistiche della tabella 5.1.

I messaggi contati durante l'esecuzione di una transazione distribuita sono le

	Portal	Auth x2	Proc	TOTALE
Senza algoritmo				
Dynamo - Dynamo (esterni)	4	4	2	**10**
Dynamo - VC	-	-	-	**-**
Dynamo - VC (esterni)	-	-	-	**-**
VC - VC (esterni)	-	-	-	**-**
Totale	*4*	*4*	*2*	***10***
Con algoritmo				
Dynamo - Dynamo (esterni)	4	4	2	**10**
Dynamo - VC	20	84	40	**144**
Dynamo - VC (esterni)	16	12	14	**42**
VC - VC (esterni)	28	-	14	**42**
Totale	*68*	*100*	*70*	***238***

Tabella 5.1: Numero di messaggi SOAP nel nostro esempio.

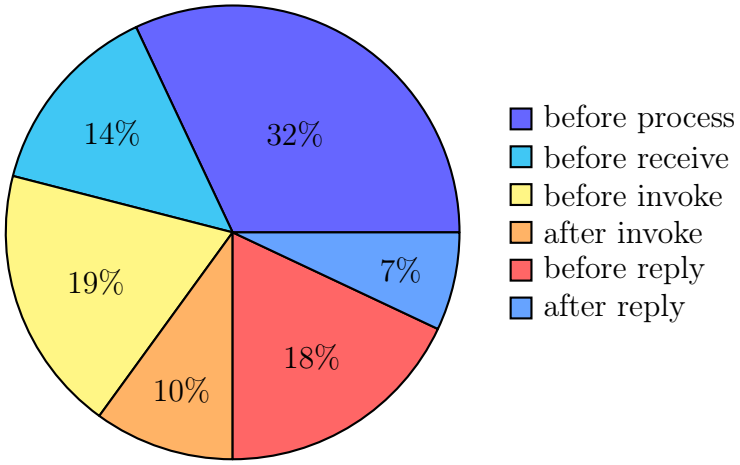
coppie di messaggi in uscita e in ingresso da ogni nodo del sistema, Portal, Auth (eseguito 2 volte) e Proc. Quindi le colonne indicano il nodo su cui stiamo contando i messaggi, e le righe indicano i componenti generici coinvolti nello scambio di messaggi.
Nel caso "senza algoritmo", si contano i messaggi corrispondenti alle coppie <invoke>/<receive> di un determinato processo BPEL. Nel caso "con algoritmo" si contano i messaggi corrispondenti alle coppie request/receive che interessano il componente (interceptor o VersionConsistencyService) di un determinato nodo.
Nei prossimi grafi analizziamo i dati raccolti.



I dati raccolti mettono in evidenza il fatto che solo il 4% dei messaggi non dipende dall'implementazione dell'algoritmo. Effettivamente questo non denota grande efficienza, soprattutto se calassimo la nostra soluzione in sistemi di grandi dimensioni, con parecchie transazioni concorrenti che impegnano i componenti.
Per migliorare la situazione si potrebbero fare degli interventi di ottimizzazione del codice, come vedremo nei prossimi grafici, oppure provare ad implementare una versione alternativa dell'algoritmo di management della version consistency, già mostrato nel paper [10]. Questa versione alternativa si chiama *On-demand set up* e prevede l'utilizzo degli archi dinamici solo al momento di richiesta di update; per cui non è necessario sovraccaricare la computazione fino a quell'istante temporale. Il problema è che la configurazione dinamica deve essere costruita bloccando l'esecuzione del sistema; in più, nel nostro contesto BPEL, sarebbe necessario ricostruire la sequenza di chiamate precedenti e questo si tradurrebbe nel posticipare la fase di inizializzazione/progress senza grossi benefici in termini di disruption.
Questo tipo di approccio può risultare utile in sistemi che hanno poche richieste di aggiornamento, mentre il nostro sistema si rivela funzionale in caso di sistemi in continua evoluzione che necessitano parecchie modifiche. Ancora, l'implementazione fatta da noi si basa solo ed esclusivamente su web services, e per gestire le informazioni degli archi non facciamo uso di database interni: questo vuol dire sì maggiore overhead legato alla rete ed ai messaggi SOAP, ma efficienza e semplicità di installazione locale, grazie all'indipendenza da un DB dedicato.
Attraverso le prossime due tabelle, che mettono in risalto i punti dell'implementazione che richiedono tanti scambi di messaggi, possiamo capire meglio come e dove intervenire per migliorare la situazione.

	Dynamo - VC	Dynamo - VC (esterni)	VC - VC (esterni)	*TOTALE*
before process	48	-	24	**72**
before receive	26	6	-	**32**
before invoke	20	24	-	**44**
after invoke	16	-	6	**22**
before reply	30	12	-	**42**
after reply	4	-	12	**16**

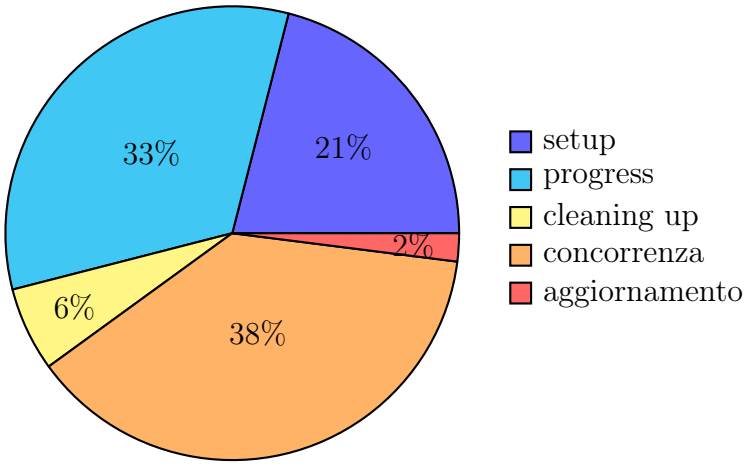
Tabella 5.2: Numero di messaggi SOAP suddivisi per istante di intercettamento.



Dal grafico si nota che la maggior parte dei messaggi avviene in corrispondenza dei punti di intercettamento identificati con before, cioè negli istanti di tempo appena prima di una BPEL activity: solo il 17% dei messaggi implementativi è invece usata dopo.
Un possibile intervento potrebbe essere quello di ritardare, laddove è possibile, alcune chiamate tra i componenti del sistema; questo permetterebbe di ridurre le latenze introdotte prima di ogni activity, e quindi all'engine di continuare la propria esecuzione, mentre l'interceptor e il web service continuerebbero a lavorare senza bloccare l'intera esecuzione.
La fase di inizializzazione, che avviene in corrispondenza del before start process occupa anch'essa una buona porzione di messaggi: ben il 32% . Questa però è l'unica porzione di messaggi su cui non è possibile avere miglioramenti, in quanto le operazioni di inizializzazione sono necessariamente effettuate prima che vengano istanziati ed eseguiti i vari processi bpel: questo perchè gli archi future della fase di set up devono essere creati prima che la transazione root cominci la propria esecuzione, e l'unico modo di avere informazioni statiche su cui costruire la configurazione dinamica iniziale è quello di fare parsing sui file *.bpel* e *.pdd*.

	Dynamo - VC	Dynamo - VC (esterni)	VC - VC (esterni)	*TOTALE*
set up	18	-	30	**48**
progress	52	24	-	**76**
cleaning up	2	-	12	**14**
concorrenza	68	18	-	**86**
aggiornamento	4	-	-	**4**

Tabella 5.3: Numero di messaggi SOAP suddivisi per categoria del metodo.



Da questi dati possiamo notare quando incidano i metodi usati per gestire la concorrenza: se calassimo la nostra soluzione in un sistema dove siamo sicuri che i client possano accedere alle risorse solo in modo esclusivo, potremmo ridurre l'overhead ben del 38%. Ottimizzando il codice, laddove sia possibile, potremmo invece ridurre il carico di messaggi legati all'algoritmo: le categorie setup, progress e cleaning up incidono per il 60%.
Infine si può notare che l'aggiornamento vero e proprio è stato implementato in modo efficiente, in quanto i messaggi usati sono solo il 2% dell'intera implementazione: c'è anche da dire però che per il momento, l'aggiornamento è solo su due versioni: aggiungendo funzionalità per gestire l'update automatico di più di due versioni probabilmente si avrebbe maggiore overhead.

5.4 Statistiche sui tempi di aggiornamento

I parametri di fondamentale importanza sull'efficienza della riconfigurazione dinamica sono il tempo di aggiornamento (*timeliness*) e il tempo di interruzione del servizio (*disruption*). L'obiettivo di questa sezione è di misurare timeliness e

disruption nel nostro sistema, validando l'implementazione della Version Consistency (anche rispetto alla Quiescence) con il criterio di aggiornamento di Concurrent Versions (rispetto a Waiting for freeness). Inoltre valuteremo l'impatto di diversi livelli di network latency sul nostro sistema, così come l'impatto del numero di transazioni concorrentemente eseguite.
L'approccio basato su tranquillity non è stato incluso nei nostri test perchè assicura solamente consistenza locale, mentre i due approcci confontati assicurano consistenza globale (come già mostrato nel capitolo 3). Per quanto riguarda il criterio di aggiornamento abbiamo tralasciato quello di Blocking for Freeness poichè, nella nostra implementazione, non è performante rispetto a Waiting for Freeness.

Nel primo esperimento abbiamo eseguito più volte la transazione distribuita del nostro esempio, utilizzando come parametro il tempo di esecuzione del web service Auth_WS che implementa la logica applicativa. Questo ci ha permesso di capire se l'esecuzione dell'algoritmo, che a prima vista sembrerebbe appesantire il processo BPEL distribuito con un numero piuttosto elevato di messaggi SOAP, ha effettivamente un impatto negativo sulle prestazioni.
Inoltre abbiamo introdotto del ritardo costante ad ogni activity ($0.5s$), sia nell'esecuzione del processo senza l'algoritmo, sia nell'esecuzione dello stesso con l'algoritmo attivo. Questo per evidenziare meglio i risultati dei nostri esperimenti e per raccogliere delle statistiche più verosimili in un contesto reale.
In questa simulazione abbiamo raccolto le seguenti statistiche, risultato della media dei tempi di più esecuzioni per un dato valore del parametro "ritardo Auth_WS".
Il ritardo di Auth_WS è misurato sull'intera transazione distribuita, questo si-

Ritardo Auth_WS [s]	**Timeliness [s]**	**Disruption [s]**
0	**5.31**	**1.70**
1	**6.25**	**1.65**
2	**7.30**	**1.69**
3	**8.31**	**1.66**
4	**9.35**	**1.70**
5	**10.28**	**1.71**
6	**11.33**	**1.70**

Tabella 5.4: Statistiche sui tempi di aggiornamento e interruzione di servizio in relazione al ritardo del web service Auth_WS.

gnifica che dobbiamo considerare la sua esecuzione due volte, infatti nel nostro esempio Auth lo chiama due volte, prima per restituire il token a Portal e poi per validarlo a Proc. La richiesta di aggiornamento arriva, per nostra convenzione, nel momento in cui Auth esegue la sua prima transazione; per questo motivo

la **timeliness** è calcolata come il tempo che intercorre da quando Auth parte a quando raggiunge la freeness, che nel nostro caso succede appena dopo la terminazione della seconda istanza di Auth, precisamente in concomitanza dell'activity di <reply>, nella quale vengono eliminati gli autoanelli dalla configurazione dinamica.
Infine la **disruption** è misurata come la perdita di working time per le transazioni rispetto all'esecuzione delle stesse senza l'applicazione dell'algoritmo, quindi, nel nostro caso si calcola come differenza tra l'esecuzione di Portal con l'algoritmo attivo e l'esecuzione di Portal senza considerare tutte le operazioni eseguite dal VersionConsistencyService per la gestione degli archi.
Le considerazioni in merito a questi risultati nascono dalla figura 5.12.
Come si può intuire facilmente anche senza i risultati sperimentali, la disruption

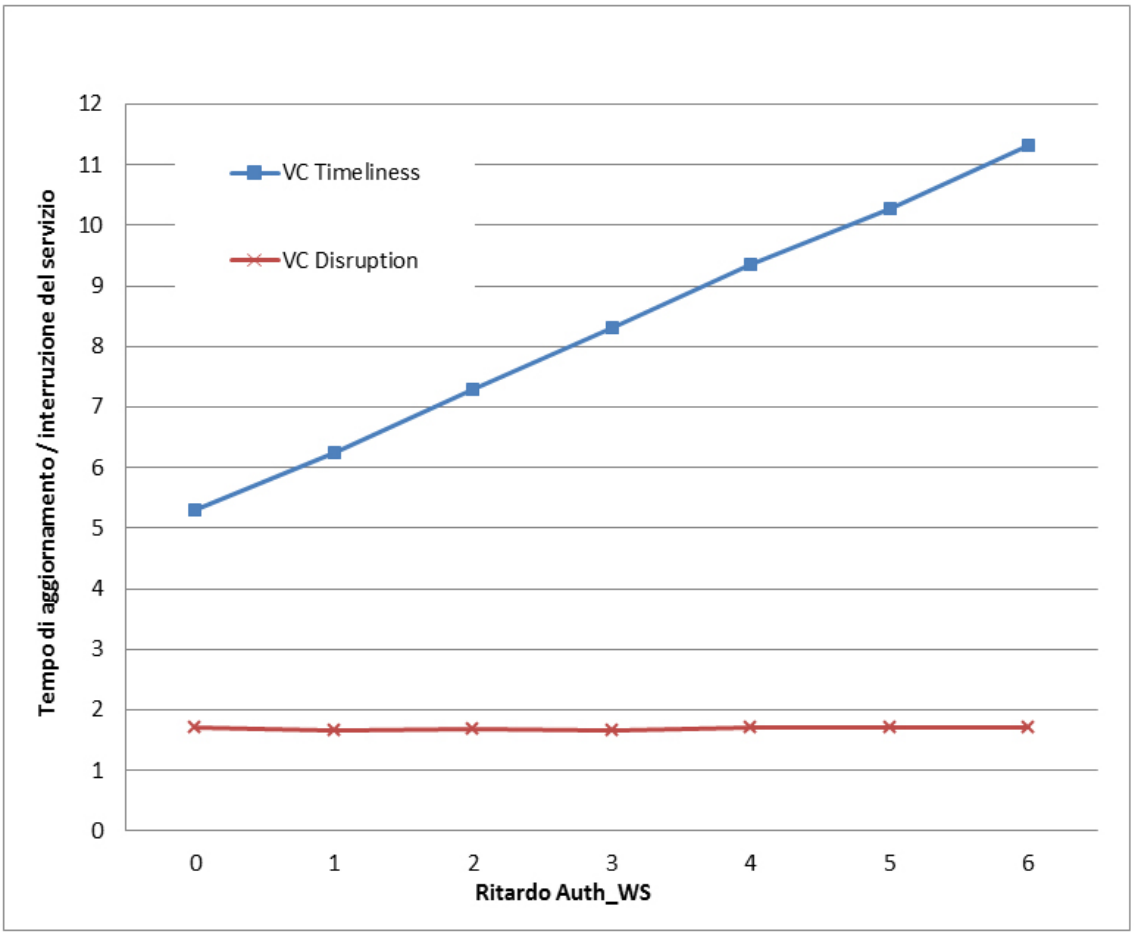


Figura 5.12: Timeliness e Disruption non sono influenzate dal ritardo di Auth_WS.

effettivamente misurata è, in media, praticamente costante e non influenzata dal ritardo del web service di Auth; infatti questo tipo di ritardo è da considerarsi

parte del tempo di esecuzione delle transazioni di Auth, per cui, anche se incrementiamo il suo valore e di conseguenza anche il tempo di esecuzione dell'intera transazione distribuita, l'algoritmo non deve costruire archi dinamici in più rispetto all'esecuzione "liscia" senza archi dinamici.
Questo si traduce in una considerazione che è interessante notare: la disruption, essendo costante, all'aumentare del tempo di esecuzione incide sempre meno sui ritardi di esecuzione totali. Ad esempio, se consideriamo una transazione root di 10.3 secondi (il caso in cui il ritardo di Auth_WS è nullo), la disruption incide per il 16%, mentre se consideriamo una transazione root di 16.3 secondi (il caso in cui il ritardo di Auth_WS è di 6s), la disruption, e dunque l'algoritmo, incide solo per il 10% dell'intera esecuzione.
Per quanto riguarda la timeliness, essa è ovviamente influenzata dal ritardo del web service, ma il suo incremento è lineare rispetto ad esso; questo ci permette di concludere che la timeliness è influenzata solo dal ritardo di Auth_WS e non dall'implementazione dell'algoritmo, nonostante il numero elevato di messaggi SOAP utilizzati per gestire gli archi dinamici. Questo rafforza quanto notato per la disruption e mette in luce la validità della nostra implementazione.

La seconda simulazione confronta Version Consistency e Quiescence valutando l'impatto della latenza della rete del nostro sistema distribuito. In questo esperimento abbiamo calcolato timeliness e disruption in funzione del parametro "delay message", che rappresenta il ritardo dovuto ai messaggi scambiati tra i vari processi BPEL. Questo ritardo è stato modellato con l'aggiunta di tempi di attesa nell'interceptor, nei punti di intercettazione before invoke e before reply: nell'esecuzione dell'intera transazione distribuita del nostro esempio questi ritardi avvengono 8 volte.
L'approccio basato su Quiescence è stato implementato sfruttando l'interceptor e il sistema che abbiamo sviluppato per la Version Consistency: abbiamo eliminato il calcolo computazionale degli archi della Version Consistency e valutato timeliness e disruption semplicemente passivando tutti i componenti che dipendono dal nodo da aggiornare (Auth). In questo modo la timeliness è calcolata sempre come la differenza tra l'istante in cui il componente è aggiornabile e quello in cui arriva la richiesta di aggiornamento; in questo caso, l'istante in cui Auth raggiunge lo stato di quiescent avviene appena Portal termina la propria esecuzione, poichè esso è l'ultimo nodo passivato che richiede servizio ad Auth.
Per quanto riguarda la disruption nel caso della Quiescence, è calcolata come la differenza tra l'esecuzione totale del processo BPEL e l'istante in cui Auth termina la propria esecuzione (seconda istanza dell'esempio): infatti questo lasso di tempo corrisponde al tempo di interruzione di servizio, il tempo cioè in cui viene attuata la passivazione di tutti i nodi dipendenti da Auth (Portal e Proc).
Disruption e timeliness nel caso della Version Consistency sono ovviamente calcolati allo stesso modo dell'esempio precedente. Le statistiche raccolte sono inserite nella tebella successiva. Come si può notare anche in figura 5.13, quando si

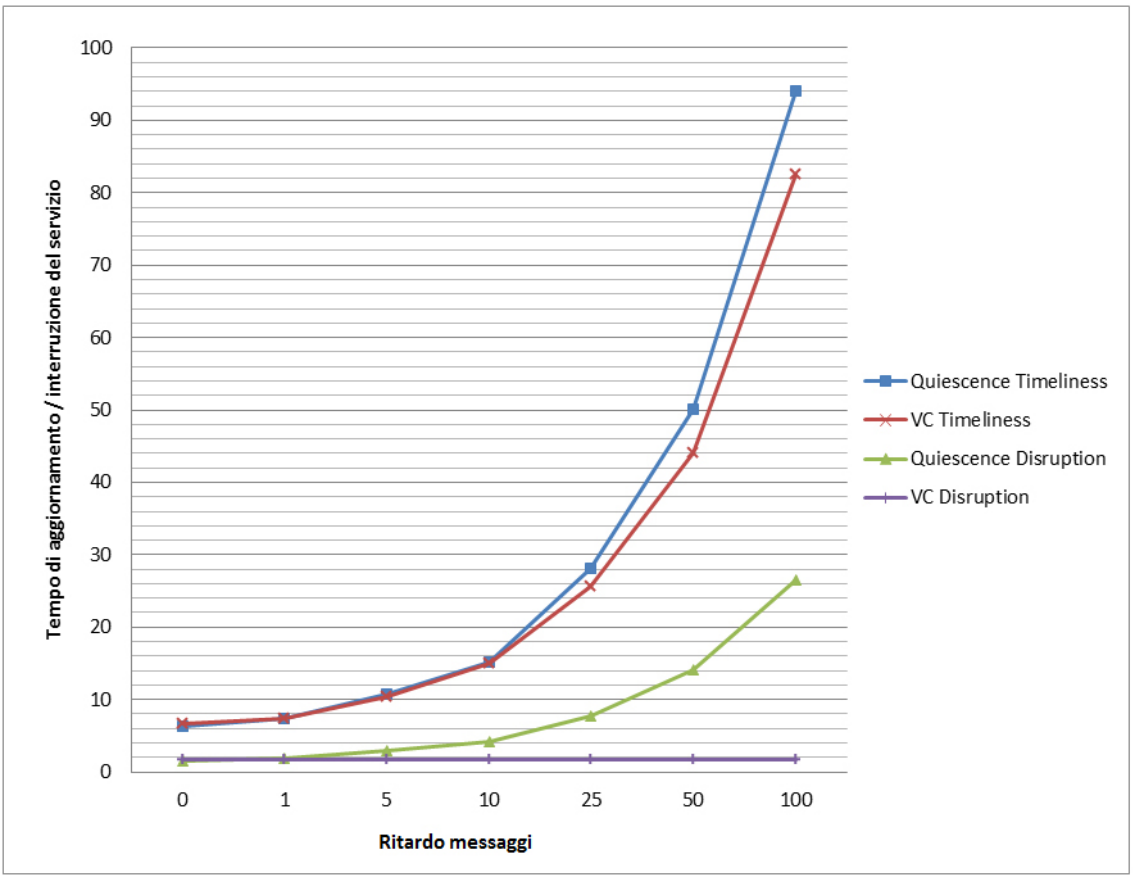


Figura 5.13: Impatto delle latenze sulla rete.

Ritardo rete [s]	Timel. VC-CV [s]	Timel. Quiesc. [s]	Disr. VC-CV [s]	Disr. Quiesc. [s]
0	**6.6**	**6.4**	**1.7**	**1.6**
1	**7.4**	**7.3**	**1.7**	**1.8**
5	**10.4**	**10.7**	**1.7**	**2.9**
10	**15**	**15.2**	**1.7**	**4.1**
25	**25.7**	**28.5**	**1.7**	**7.8**
50	**44.1**	**50.1**	**1.7**	**14.1**
100	**82.5**	**94.0**	**1.7**	**26.6**

Tabella 5.5: Timeliness e disruption in relazione al ritardo della rete.

utilizza il criterio di Version Consistency, la disruption è costante, in quanto il ritardo della rete non influisce sulla creazione/rimozione degli archi. Invece nel caso della Quiescence, il tempo di interruzione di servizio è influenzato dal ritardo con cui i messaggi sono scambiati, infatti la passivazione di Portal e Proc è ovviamente ritardata.
La disruption è leggermente inferiore utilizzando la Quiescence se non ci sono ritardi sulla rete, poichè questo criterio non deve gestire archi o logica applicativa particolarmente complessa; aumentando il delay dei messaggi, calando così la soluzione in un modello più realistico, la differenza tra le due soluzioni è però evidente.
Per quanto riguarda la timeliness, le due soluzioni sono più simili, anche se per alti ritardi della rete la Version Consistency è comunque più performante.

L'ultima simulazione è stata fatta per analizzare le differenti strategie per il raggiungimento della freeness, in particolare per verificare se implementare il criterio di riconfigurazione *Waiting for Freeness* è effettivamente meno efficiente rispetto a quello di *Concurrent Versions*. Intuitivamente la timeliness della strategia WF dovrebbe essere altamente sensibile all'aumento delle transazioni concorrenti sul nostro sistema.
Il nostro esperimento è eseguito modellando dei ritardi casuali (distribuiti uniformemente in un range opportuno) nella rete durante lo scambio di messaggi tra i vari componenti del CDBS, cioè ritardando casualmente le varie activity dell'intera transazione BPEL distribuita, con lo scopo di calare il nostro esempio in un contesto reale.
La timeliness è calcolata eseguendo la transazione distribuita più volte e facendo la media di più esecuzioni per un determinato numero di transazioni concorrenti. La timeliness è il tempo che intercorre dalla richiesta di aggiornamento (per noi coincide con l'inizio della prima transazione su Auth) a quando il nodo Auth raggiunge la freeness: in questo scenario, se applichiamo il criterio WF, la freeness è raggiunta solo quando il nodo Auth è free rispetto a tutte le transazioni distribuite che stanno eseguendo sul sistema; se invece applichiamo il criterio CV, la

freeness è sempre raggiunta quando il nodo è free rispetto la prima transazione distribuita, infatti tutte le altre transazioni continueranno sulla vecchia versione di Auth.
Ecco le **medie** dei dati raccolti. Come si nota da questi dati, rappresentati in

Numero transazioni	Timeliness VC-CV [s]	Timeliness VC-WF [s]
1	**6.0**	**6.0**
2	**6.0**	**7.7**
3	**6.0**	**12.6**
4	**6.0**	**16.7**
5	**6.0**	**21.7**
7	**6.0**	**30.0**
10	**6.0**	**47.2**

Tabella 5.6: Il tempo di aggiornamento nel nostro sistema con i criteri di Concurrent Versions (CV) e Waiting for Freeness (WF).

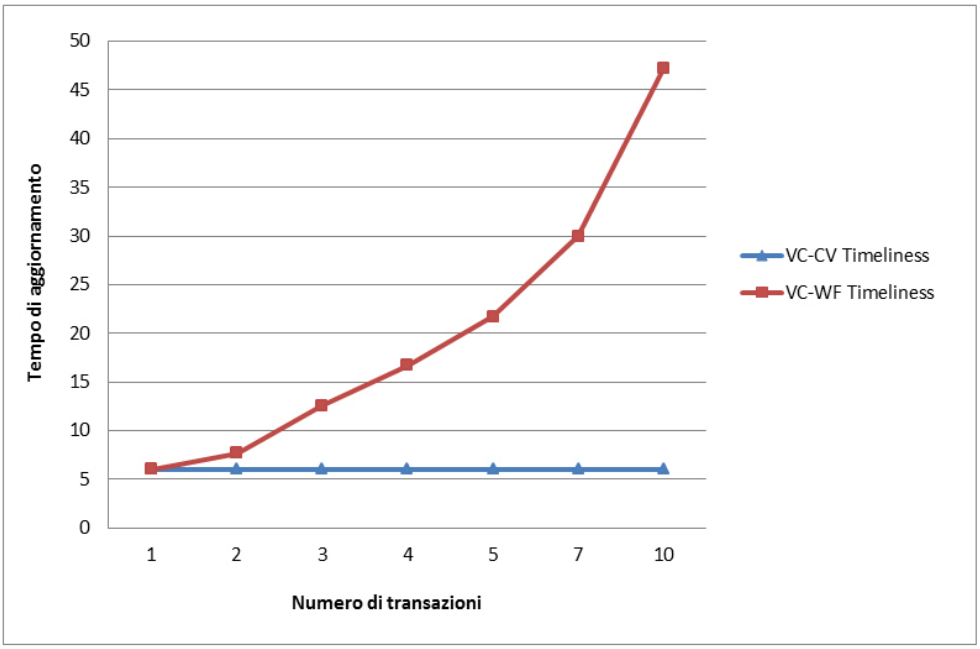


Figura 5.14: I tempi di aggiornamento premiano la strategia di Concurrent Versions.

figura 5.14, il risultato è evidente: all'aumentare delle transazioni concorrenti, il criterio CV non fa mai aumentare il tempo di aggiornamento, mentre il criterio WF sì. Con 10 transazioni attive c'è una differenza di 41 secondi che equivale ad un ritardo 7 volte superiore per la strategia WF rispetto alla strategia CV!

Capitolo 6

Conclusioni

6.1 Contributi del lavoro

L'obiettivo del nostro progetto di tesi è stato quello di realizzare un framework per la riconfigurazione dinamica dei processi di business.
La motivazione di questa scelta è nata dalla fusione di due aspetti:

1. l'assenza di software che gestisca la consistenza degli aggiornamenti di questo tipo di processi, particolarmente soggetti a rapidi cambiamenti nei requisiti che devono soddisfare;

2. la necessità di calare il criterio di Version Consistency in un contesto reale, per valutarne l'effettiva efficacia.

L'obiettivo è stato raggiunto grazie all'interceptor di Dynamo e al web service di gestione della Version Consistency, che compongono il nostro framework: con questi due strumenti è possibile effettuare aggiornamenti a runtime garantendo la consistenza dell'intero sistema e lo svolgimento sicuro di tutte le transazioni in corso.
La scelta di lavorare su CBDS composti da processi BPEL è stata mirata, poiché la suddivisione sequenziale e ben definita delle attività prevista dal linguaggio permette di intercettare le transazioni in istanti precisi che ben si prestano per l'esecuzione dell'algoritmo. É stato quindi possibile gestire la realizzazione del criterio di Version Consistency eseguendo nel modo corretto e negli istanti giusti la logica di business. Di conseguenza ogni componente del sistema è sempre consapevole del fatto di potersi aggiornare o meno, in base al proprio stato di *freeness* determinato localmente.
Ovviamente non è stato possibile limitarsi all'implementazione dell'algoritmo, il VersionConsistencyService esegue logiche molto più complesse per gestire la memorizzazione della configurazione dinamica o di parametri indispensabili come il process id della transazione di root o il nome dei processi invocanti. Queste ultime informazioni non si possono infatti ricavare tramite l'interceptor di Dynamo:

è quindi stato necessario integrare i metodi dell'algoritmo con delle funzionalità aggiuntive che gestiscono la loro propagazione nel corso delle varie transazioni.

L'implementazione realizzata permette di aggiornare i componenti sia con la strategia *Waiting for Freeness* che con la strategia *Concurrent Versions*. Tra le due possibilità è preferibile la seconda poichè offre performance migliori per quanto riguarda i tempi di aggiornamento, consentendo di mantenere in contemporanea due versioni dello stesso componente e reindirizzando le transazioni in esecuzione sulla versione corretta.

Le simulazioni eseguite sul nostro sistema hanno evidenziato buone performances per quanto riguarda la latenza sul tempo di aggiornamento (*timeliness*) e il tempo di interruzione del servizio (*disruption*). Il numero di messaggi scambiati tra i vari componenti del sistema distribuito è prevedibilmente superiore attivando il nostro framework. Lo scambio di messaggi è fondamentale per gestire l'esecuzione del criterio di Version Consistency, per questo motivo l'applicazione dell'algoritmo introduce un certo overhead sulla transazione distribuita. Tuttavia i nostri test mettono in evidenza che l'overhead introdotto sulla rete del sistema non aumenta proporzionalmente al tempo di esecuzione della transazione di business, quindi soprattutto per transazioni long-running eseguite in concorrenza, il tempo di esecuzione dell'algoritmo risulterebbe trascurabile rispetto al tempo totale. Del resto sarebbe impensabile realizzare questo tipo di aggiornamenti senza introdurre alcun tipo di overhead sul sistema.
I dati sperimentali evidenziano anche performances migliori rispetto ad un'eventuale implementazione basata sul criterio di Quiescence. Alcuni test eseguiti sul sistema mostrano risultati migliori per quanto riguarda timeliness e disruption. La scelta di utilizzare la Version Consistency come base del nostro framework ci ha quindi consentito di effettuare aggiornamenti sicuri ma più rapidi, per cui è la scelta attualmente più performante possibile.

6.2 Sviluppi futuri

Per diminuire l'overhead e il numero di messaggi che vengono scambiati per la gestione dell'algoritmo, e quindi apportare dei miglioramenti alla nostra soluzione, è stata effettuata un'analisi che ha messo in evidenza alcuni possibili sviluppi futuri. Li elenchiamo brevemente:

- **Ottimizzazione della gestione della concorrenza:** l'attuale gestione della propagazione del process id della root transaction (descritta ampiamente nel paragrafo 4.3.5) consente di gestire al meglio la sincronizzazione in caso di transazioni concorrenti, ma risulta essere abbastanza pesante in quanto ad overhead. Una prima idea potrebbe essere la modifica dell'interceptor in modo da aggiungere questo dato tra i parametri del messaggio

SOAP scambiato da due nodi nel corso di una comunicazione. In questo caso il PID verrebbe propagato con più semplicità rispetto ad ora.
Un'altra soluzione potrebbe essere lo spostamento dell'ArrayList che contiene le stringhe di `rootPid_hash` all'interno dell'interceptor di Dynamo. Allo stato attuale questa informazione si trova sul VersionConsistencyService; un nodo per conoscere il process id della root transaction o il nome del proprio processo invocante deve quindi effettuare delle chiamate che comportano overhead. Noi abbiamo preferito mantenere l'ArrayList sul web service esterno per poter gestire l'intero criterio di Version Consistency in modo totalmente decentralizzato. Tale scelta consente una maggiore portabilità, permettendo, ad esempio, di riutilizzare la nostra soluzione in un altro ambito, magari discostandosi dai CBDS basati unicamente su componenti BPEL.

- **Supporto di processi BPEL complessi:** al momento il nostro interceptor gestisce l'esecuzione e l'intercettamento di processi BPEL con sequenze di <Receive>, <Assign>, <Invoke> e <Reply>. Un processo di business si può rendere molto più complesso sfruttando altri costrutti come ad esempio <Switch>, <While>, <Flow>, <Pick>. Aggiungendo la gestione di questi costrutti nell'interceptor sarebbe possibile gestire sistemi distribuiti basati su componenti BPEL che svolgono transazioni dotate di una logica molto più complessa.

- **Funzionalità accessorie per l'aggiornamento:** al momento all'interno della nostra implementazione la richiesta di aggiornamento avviene automaticamente fin dall'inizio della prima istanza del nodo da aggiornare. Quindi non appena questo componente acquisisce stato di `free`, in base al criterio di Version Consistency, esso viene aggiornato modificando la logica di business eseguita dal web service che gestisce la logica stessa della transazione. Questo approccio è stato utile per eseguire il nostro scenario e per verificare il corretto funzionamento del framework. In futuro sarà possibile garantire maggiore libertà all'utente facendo in modo che il framework gestisca in modo automatico più di due versioni del web service del nodo da aggiornare, a patto che il tutto sia stato configurato in modo opportuno. Inoltre sarebbe utile sviluppare un client che dia la possibilità all'utente di scegliere arbitrariamente l'istante in cui far pervenire al sistema la richiesta di aggiornamento per un componente.

In conclusione,è sicuramente possibile migliorare l'implementazione del framework per la gestione della Version Consistency introducendo alcune delle idee mostrate sopra. Tuttavia ciò che è stato sviluppato costituisce già un'ottima base per effettuare aggiornamenti automatici a runtime dei processi di business garantendo la correttezza del sistema globale.

Bibliografia

[1] J.Appavoo D.Da Silva O.Krieger R.W.Wiesniewski J.Kerr A.Baumann, G.Heiser. Providing dynamic update in an operating system. *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference, pages 32-32*, 2005.

[2] K. N. Biyani and S. S. Kulkarni. Assurance of dynamic adaptation in distributed systems. *J. Parallel Distrib. Comput*, 2008.

[3] T. Saridakis C. Bidan, V. Issarny and A. Zarras. A dynamic reconfiguration service for corba. *CDS '98: Proceeding of the International Conference on Configurable Distributed Systems*, 1998.

[4] X. Chen and M. Simons. A component framework for dynamic reconfiguration of distributed systems. *CD '02: Proceeding of the IFIP/ACM Working Conference on Component Deployment, pages 82-96*, 2002.

[5] M. Goedicke G. Taentzer and T. Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. *TAGT '98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations, pages 179-193*, 2000.

[6] J. S. Foster I. Neamtiu, M. Hicks and P. Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 37-49*, 2008.

[7] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 1990.

[8] A. Lopes M. Wermellinger and J. L. Fiadeiro. A graph based architectural (re)configuration language. *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, pages 21-32.*

[9] L.Shira S. Ajamani, B.Liskov. Modular software upgrade for distributed systems. *ECOOP'06: Proceedings of the 20th European conference on Object-Oriented Programming , pages 452-476*, 2006.

[10] C.Ghezzi V.Panzica La Manna J.Lu X.Ma, L.Baresi. Version-consistent dynamic reconfiguration of component-based distributed systems. *ESEC/FSE '11: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011.

[11] Y. Berbers Y. Vandewoude, P. Ebraert and T. D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Transactions on Software Engineering*, 2007.

[12] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. *ICSE '06*, 2006.

Elenco delle figure

Elenco delle tabelle

Appendice A

Algorithm of dynamic dependence management

use $f(T)$	▷ out-going edges might be used in the future by T
use $p(T)$	▷ out-going static edges used in the past by T
use $localTxs()$	▷ on-going local transactions
const THIS	▷ this node
var $OES \leftarrow \{\}$	▷ dynamic edges leaving from this node
var $IES \leftarrow \{\}$	▷ dynamic edges arriving at this node

Algorithm 1 Setting up

1: *Upon being initiated a root tx T:*
2: **let** $lfe = THIS \xrightarrow[T]{future} THIS$, $lpe = THIS \xrightarrow[T]{past} THIS$
3: $OES \leftarrow OES \cup \{lfe, lpe\}$, $IES \leftarrow IES \cup \{lfe, lpe\}$
4:
5: *Upon a root Tx T initiating its first sub-tx:*
6: **for all** $ose = THIS \xrightarrow{static} C \in f(T)$ **do**
7: **let** $fe = THIS \xrightarrow[T]{future} C$
8: $OES \leftarrow OES \cup \{fe\}$
9: NOTIFY-FUTURE-CREATE fe
10: wait for ACK-FUTURE-CREATE fe
11: **end for**
12:
13: *Upon receiving* NOTIFY-FUTURE-CREATE $xe = C \xrightarrow[T]{future} THIS$
14: $IES \leftarrow IES \cup \{xe\}$
15: **for all** $C' \in \{X | THIS \xrightarrow{static} X\}$ **do**
16: **let** $fe = THIS \xrightarrow[T]{future} C'$
17: **if** $fe \notin$ OES **then**
18: $OES \leftarrow OES \cup \{fe\}$
19: NOTIFY-FUTURE-CREATE fe
20: wait for ACK-FUTURE-CREATE fe
21: **end if**
22: **end for**
23: ACK-FUTURE-CREATE xe

Algorithm 2 Progressing

1: **procedure** RemoveFutureEdges(T) ▷ remove T-labelled future edges without breaking future-validity
2: **for all** $fe \in OES : fe = THIS \xrightarrow[T]{future} C \wedge C \neq THIS$ **do**
3: **if** $(\nexists e \in IES : e = C' \xrightarrow[T]{future} THIS \wedge C' \neq THIS) \wedge (\nexists T' \in localTxs() : root(T') = T \wedge THIS \xrightarrow{static} C \in f(T'))$ **then**
4: $OES \leftarrow OES - \{fe\}$
5: NOTIFY-FUTURE-REMOVEfe
6: **end if**
7: **end for**
8: **end procedure**
9:
10: *Upon being initiated a sub-tx T via $se = C \xrightarrow{static} THIS$:*
11: **let** $lfe = THIS \xrightarrow[root(T)]{future} THIS$, $lpe = THIS \xrightarrow[root(T)]{past} THIS$
12: $OES \leftarrow OES \cup \{lfe, lpe\}$, $IES \leftarrow IES \cup \{lfe, lpe\}$
13: ACK-SUBTX-INIT se $root(T)$
14:
15: *Upon receiving* ACK-SUBTX-INIT T *via* $se = THIS \xrightarrow{static} C$:
16: RemoveFutureEdges(T)
17:
18: *Upon receiving* NOTIFY-FUTURE-REMOVE $fe = C \xrightarrow[T]{future} THIS$:
19: $IES \leftarrow IES - \{fe\}$
20: RemoveFutureEdges(T)
21:
22: *Upon ending a sub-tx T initiated via $se = C \xrightarrow{static} THIS$:*
23: NOTIFY-SUBTX-END se $root(T)$
24:
25: *Upon receiving* NOTIFY-SUBTX-END T *via* $se = THIS \xrightarrow{static} C$:
26: **let** $pe = THIS \xrightarrow[T]{past} C$
27: $OES \leftarrow OES \cup \{pe\}$
28: NOTIFY-PAST-CREATE pe
29:
30: *Upon receiving* NOTIFY-PAST-CREATE $pe = C \xrightarrow[T]{past} THIS$:
31: $IES \leftarrow IES \cup \{pe\}$
32: **if** $\nexists T' \in localTsx() : root(T') = T$ **then**
33: **let** $lfe = THIS \xrightarrow[T]{future} THIS$, $lpe = THIS \xrightarrow[T]{past} THIS$
34: $OES \leftarrow OES - \{lfe, lpe\}$, $IES \leftarrow IES - \{lfe, lpe\}$
35: **end if**
36: RemoveFutureEdges(T)

Algorithm 3 Cleaning up

1: **procedure** RemoveAllEdges(T) ▷ remove all T-labelled edges
2: **for all** $e \in OES : e = THIS \xrightarrow[T]{*} *$ **do**
3: $OES \leftarrow OES - \{e\}$
4: **if** $e = THIS \xrightarrow[T]{future} C \wedge C \neq THIS$ **then**
5: NOTIFY-FUTURE-REMOVE e
6: **else if** $e = THIS \xrightarrow[T]{past} C \wedge C \neq THIS$ **then**
7: NOTIFY-PAST-REMOVE e
8: **end if**
9: **end for**
10: **end procedure**
11:
12: *Upon ending a root tx T:*
13: **let** $lfe = THIS \xrightarrow[T]{future} THIS$, $lpe = THIS \xrightarrow[T]{past} THIS$
14: $OES \leftarrow OES - \{lfe, lpe\}$, $IES \leftarrow IES - \{lfe, lpe\}$
15: RemoveAllEdges(T)
16:
17: *Upon receiving* NOTIFY-PAST-REMOVE $pe = C \xrightarrow[T]{past} THIS$
18: $IES \leftarrow IES - \{pe\}$
19: RemoveAllEdges(T)

Appendice B

Installazione e uso: i processi BPEL

Per definire i processi BPEL è possibile utilizzare degli appositi designer, senza scrivere a mano i file XML (*.bpel* e *.pdd*). Per la definizione dei nostri processi abbiamo scelto di utilizzare un plugin dell'ambiente Eclipse chiamato *Eclipse BPEL Designer*. Questo plugin mette a disposizione degli strumenti grafici per la progettazione delle attività e del loro flusso di esecuzione.

I processi BPEL, per essere effettivamente eseguiti, devono essere deployati su un application server dotato di un engine adeguato per la loro esecuzione. Non è sufficiente quindi progettare il processo BPEL con il designer, ma bisogna anche generare il file *.pdd* in cui vengono specificati i partner, e generare il file *WSDL* che espone le operazioni eseguite dal processo, con la specifica dei parametri di ingresso e di uscita.
Questi file andranno inseriti in un apposito archivio che verrà posizionato sul server, il quale conterrà anche i WSDL dei processi partner che verranno invocati dal processo BPEL in questione.
In seguito alla progettazione dei processi BPEL, per una maggiore comodità abbiamo spostato i file *.bpel* e i WSDL necessari per ogni archivio nell'ambiente proprietario ActiveVOS. Questo software permette di esportare con maggior facilità i processi generando l'archivio *.bpr* da posizionare sul server.
Prima di generare l'archivio *.bpr* di ogni processo bisogna generare il file *.pdd* associato al file *.bpel*. Questa operazione può essere effettuata automaticamente ma bisogna completare a mano il file inserendovi i parametri degli end point reference dei processi partner. Per maggiore chiarezza mostriamo il file Portal.pdd associato alla definizione di processo Portal.bpel:

```
<?xml version="1.0" encoding="UTF-8"?>
<pdd:process xmlns:bpelns="http://portal"
        xmlns:pdd="http://schemas.active-endpoints.com/pdd/2006/08/pdd.xsd"
        location="bpel/Portal_VC/bpelContent/Portal.bpel" name="bpelns:Portal" platform="opensource">
  <pdd:partnerLinks>
    <pdd:partnerLink name="auth_pl">
      <pdd:partnerRole endpointReference="static" invokeHandler="default:Address">
        <wsa:EndpointReference xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
              xmlns:s="http://DefaultNamespaceAuth">
            <wsa:Address>urn:authservice</wsa:Address>
```

```
                <wsa:ServiceName PortName="Auth">s:AuthService</wsa:ServiceName>
            </wsa:EndpointReference>
        </pdd:partnerRole>
    </pdd:partnerLink>
    <pdd:partnerLink name="client">
        <pdd:myRole allowedRoles="" binding="MSG" service="clientService"/>
    </pdd:partnerLink>
    <pdd:partnerLink name="proc_pl">
        <pdd:partnerRole endpointReference="static" invokeHandler="default:Address">
            <wsa:EndpointReference xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
                xmlns:s="http://DefaultNamespace">
                <wsa:Address>urn:procservice</wsa:Address>
                <wsa:ServiceName PortName="proc">s:procService</wsa:ServiceName>
            </wsa:EndpointReference>
        </pdd:partnerRole>
    </pdd:partnerLink>
  </pdd:partnerLinks>
  <pdd:references>
    <pdd:wsdl location="project:/Portal_VC/bpelContent/proc.wsdl"
                namespace="http://DefaultNamespaceProc"/>
    <pdd:wsdl location="project:/Portal_VC/bpelContent/Auth.wsdl"
                namespace="http://DefaultNamespaceAuth"/>
    <pdd:wsdl location="project:/Portal_VC/bpelContent/PortalArtifacts.wsdl"
                namespace="http://portal"/>
  </pdd:references>
</pdd:process>
```

Con ActiveVOS, quando il progetto è completo (quindi contiene il file *.bpel*, il file *.pdd*, il proprio WSDL e i WSDL dei processi partner) è possibile esportare il tutto in un *Business Process Archive File*, ovvero un archivio *.bpr*.
É stato generato un archivio *.bpr* per ogni processo *.bpel* e i vari archivi sono posizionati su differenti istanze di Tomcat. L'engine di esecuzione di tali processi, disponibile su tutti i Tomcat, è ActiveBpel come affermato in precedenza.

Per avviare una transazione sfruttiamo un'apposita funzionalità di ActiveVOS che permette di testare Web Services. ActiveVOS permette di utilizzare dei client che invocano Portal passandogli la stringa desiderata ed avviando quindi l'intera transazione distribuita. É possibile ovviamente avviare transazioni in successione oppure anche in concorrenza: ogni volta che viene inviata una stringa dal client viene creata una nuova istanza di Portal che avvia una transazione. In figura B.1 si vede come viene avviata una transazione passando in input la stringa `test`.

É possibile seguire l'intera transazione attraverso le console di ActiveBpel sui vari server. Ovviamente sulla console di un server si potrà seguire l'andamento del processo deployato sul server stesso, monitorando il corretto svolgimento delle attività, i valori delle variabili e le cause di eventuali errori. Visualizziamo ad esempio in figura B.2 la console di ActiveBpel sul server che ospita Portal.

Nella finestra *Process* è possibile visualizzare la sequenza delle attività svolte, con la tick che mette in evidenza che sono state completate correttamente. Nella scheda *Variable Instance Data* stiamo monitorando il valore della variabile *input*, che infatti è la stringa *test* ricevuta dal client.

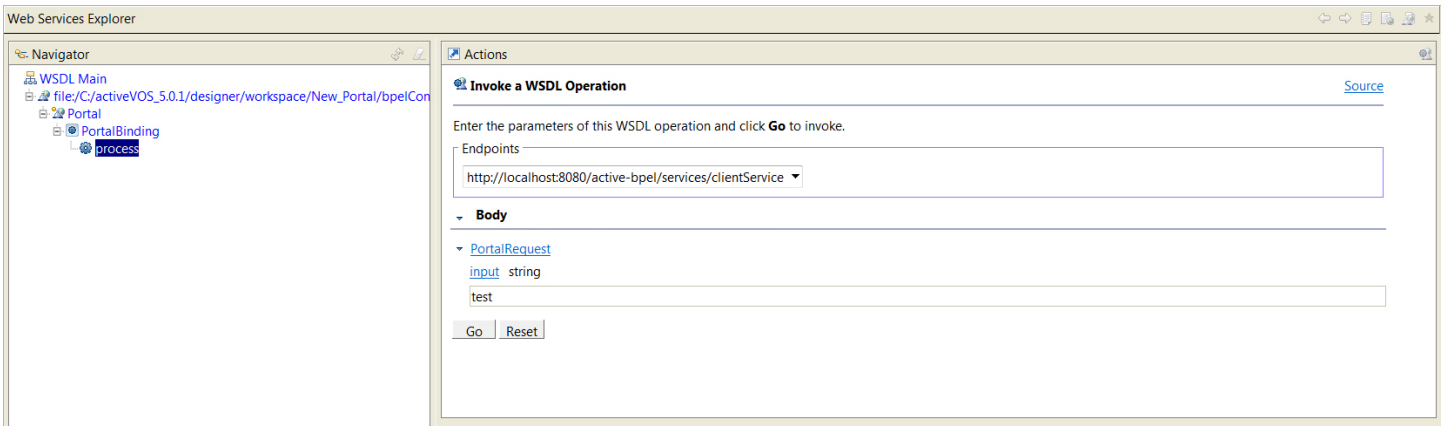


Figura B.1: Test con web service explorer di ActiveVos.

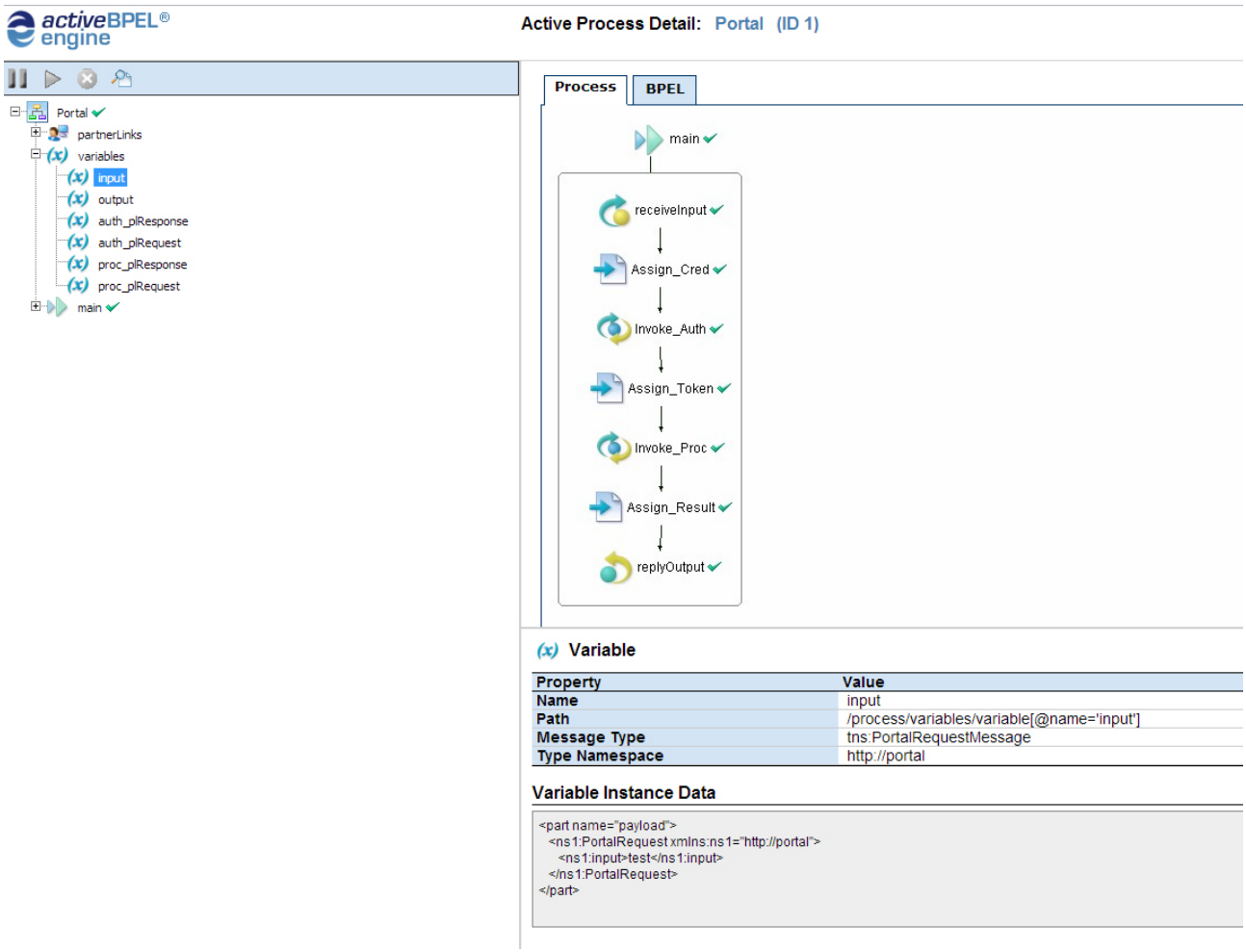


Figura B.2: Console di ActiveBpel.

Al termine dell'esecuzione, su ActiveVos (figura B.3) compare il risultato restituito al client (in questo caso `true` poiché la lunghezza della stringa è maggiore di 2), che è possibile monitorare anche attraverso la console di ActiveBpel visualizzando il valore della variabile *output* oppure visualizzando ciò che viene passato durante l'attività di <Reply>.

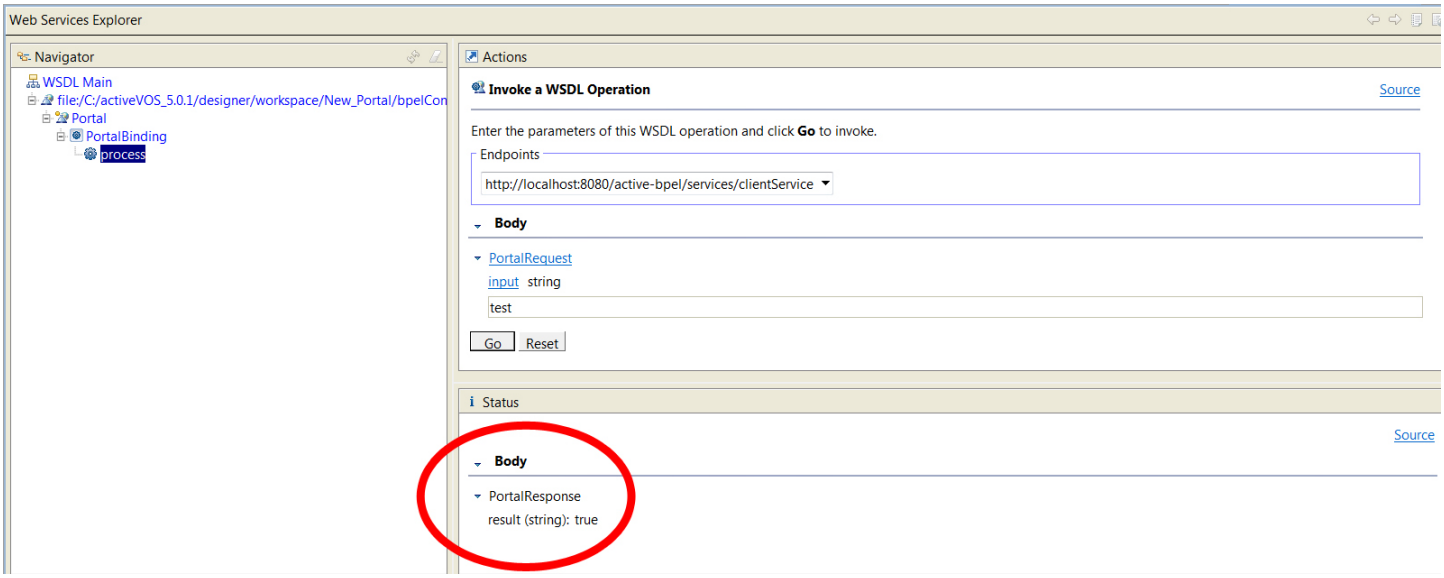


Figura B.3: Risultato della transazione distribuita.

Il web service Auth_WS viene collocato sul server Tomcat sfruttando le librerie di Apache Axis, che permettono anche la sua corretta esecuzione. Il deploy del web service su Tomcat è molto semplice, è infatti sufficiente cambiare l'estensione della classe Java in *.jws*.
A questo punto si colloca il file *.jws* in un'apposita cartella di Tomcat e nella console di Axis sarà possibile verificare la presenza del web service. Axis genera in automatico anche il WSDL del servizio, che quindi è disponibile per chiunque voglia interefacciarsi e richiedere funzionalità. Ricordiamo che il file WSDL è fondamentale per esporre le proprie operazioni e per descrivere i parametri di ingresso e di uscita. Il WSDL verrà collocato anche nell'archivio .bpr del processo BPEL Auth, visto che Auth_WS è un suo partner durante la transazione. Per maggiore chiarezza riportiamo il contenuto del file *Auth_Service.wsdl*, ovvero il file WSDL di Auth_WS:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://DefaultNamespace"
xmlns:apachesoap="http://xml.apache.org/xml-soap" xmlns:impl="http://DefaultNamespace"
xmlns:intf="http://DefaultNamespace" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.4
Built on Apr 22, 2006 (06:55:48 PDT)-->
 <wsdl:types>
  <schema elementFormDefault="qualified" targetNamespace="http://DefaultNamespace"
```

```
                xmlns="http://www.w3.org/2001/XMLSchema">
    <element name="processCred">
     <complexType>
      <sequence>
       <element name="token" type="string"/>
       <element name="cred" type="xsd:string"/>
       <element name="operation" type="xsd:int"/>
      </sequence>
     </complexType>
    </element>
    <element name="processCredResponse">
     <complexType>
      <sequence>
       <element name="processCredReturn" type="xsd:string"/>
      </sequence>
     </complexType>
    </element>
   </schema>
  </wsdl:types>
    <wsdl:message name="processCredResponse">
       <wsdl:part element="impl:processCredResponse" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="processCredRequest">
       <wsdl:part element="impl:processCred" name="parameters"/>
    </wsdl:message>
    <wsdl:portType name="Auth_Service">
       <wsdl:operation name="processCred">
          <wsdl:input message="impl:processCredRequest" name="processCredRequest"/>
          <wsdl:output message="impl:processCredResponse" name="processCredResponse"/>
       </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="Auth_ServiceSoapBinding" type="impl:Auth_Service">
       <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
       <wsdl:operation name="processCred">
          <wsdlsoap:operation soapAction=""/>
          <wsdl:input name="processCredRequest">
             <wsdlsoap:body use="literal"/>
          </wsdl:input>
          <wsdl:output name="processCredResponse">
             <wsdlsoap:body use="literal"/>
          </wsdl:output>
       </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="Auth_ServiceService">
       <wsdl:port binding="impl:Auth_ServiceSoapBinding" name="Auth_Service">
          <wsdlsoap:address location="http://localhost:8080/axis/Auth_Service.jws"/>
       </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Appendice C

Installazione e uso: il framework

Il nostro framework è composto essenzialmente da due componenti: l'interceptor di Dynamo e il VersionConsistencyService.

Per poter usufruire delle funzionalità dell'**interceptor** è necessario creare la libreria *ae_rtbpel.jar* dell'intero Dynamo. Questo passaggio è realizzabile attraverso vari procedimenti. Il più semplice è attuabile all'interno dell'ambiente Eclispe tramite l'esportazione dell'intero Project, tenendo conto delle funzionalità di AspectJ, come si vede in figura C.1.
Ricordiamo che questo è assolutamente necessario, poichè AspectJ ci permette di inserire nel codice alcuni "punti di intercettamento" che agiscono in concomitanza di determinati eventi del processo BPEL eseguito.

Per quanto riguarda il **VersionConsistencyService**, il deploy sul server avviene attraverso la creazione di un archivio *.aar* da inserire nell'apposita cartella di *Axis2* ($Catalina\_Home/webapps/axis2/WEB-INF/services$) in Tomcat. Partendo dalla classe Java che contiene i numerosi metodi che implementano l'algoritmo di Version Consistency, i passi necessari per la creazione dell'archivio sono i seguenti:

1. all'interno di Eclipse si clicca con il tasto destro sulla classe Java e si selezionano le voci del menu *Web Services -> Create web service*, in modo da generare il web service partendo dalla classe Java (tecnica *bottom-up*);

2. si configura la generazione *bottom-up* impostando Tomcat come *Server runtime* e Apache Axis2 come *Web service runtime* (figura C.2);

3. Con l'operazione precedente viene generato il file *services.xml* che andrà modificato aggiungendovi l'attributo *scope* nel tag <service>. Lo scope va impostato come *application*; ciò è fondamentale per far sì che il web service sia di tipologia **stateful** e mantenga così in memoria la struttura dati tra una richiesta e l'altra. Il web service quindi svolge il ruolo di

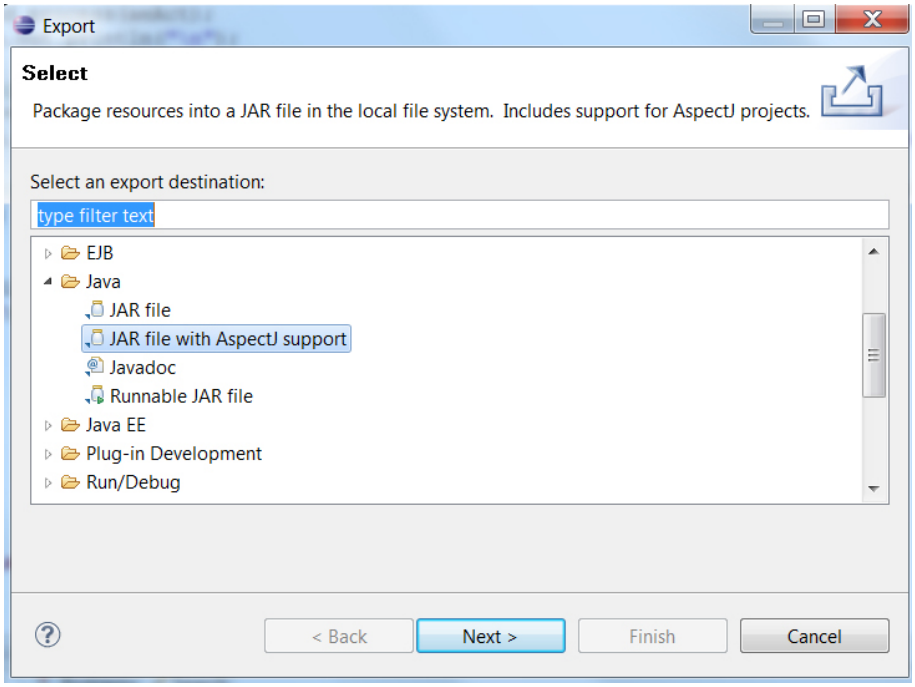


Figura C.1: Creazione dell'archivio di Dynamo in Eclipse.

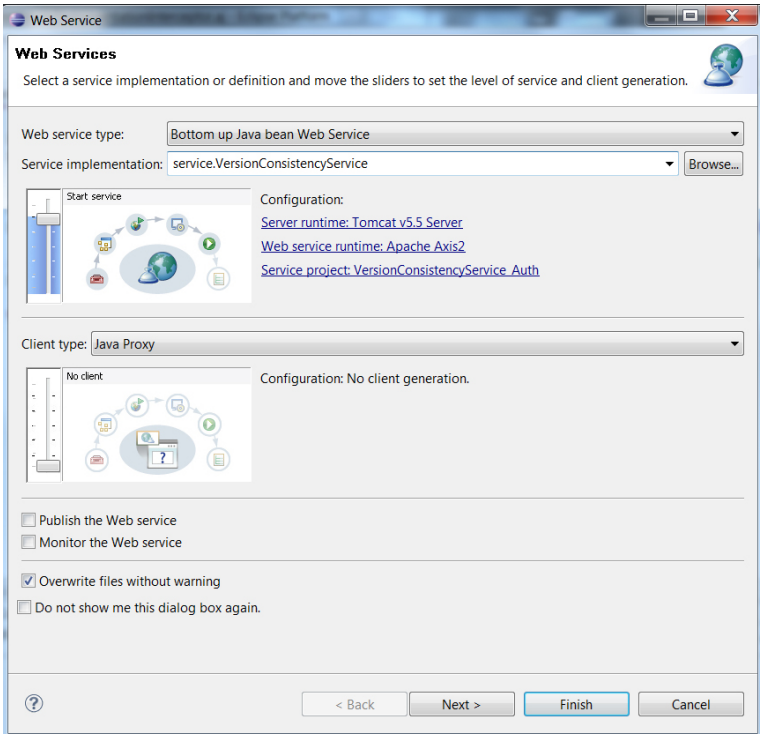


Figura C.2: Creazione web service: VersionConsistencyService.

oggetto condiviso singleton che serve tutte le richieste; ecco la struttura del file xml:

```
<service name="VersionConsistencyService" scope="application">
        <Description>
                Please Type your service description here
        </Description>
        <messageReceivers>
                <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-only"
                                 class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver" />
                <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
                                 class="org.apache.axis2.rpc.receivers.RPCMessageReceiver"/>
        </messageReceivers>
        <parameter name="ServiceClass" locked="false">service.VersionConsistencyService</parameter>
</service>
```

4. tramite il prompt dei comandi di Windows ci si posiziona nella cartella del workspace di Eclipse che contiene la classe Java del Web Service e si esegue il seguente comando per la creazione dell'archivio .aar:

```
 jar cvf VersionConsistencyService.aar VersionConsistencyService.class
VersionConsistencyService$DynEdges_Struct.class META-INF
```

5. a questo punto è possibile inserire l'archivio .aar sul server Tomcat, nell'apposita cartella di Axis2.

Riassumendo, per utilizzare la nostra architettura bisognerà inserire Dynamo e il VersionConsistencyService in ogni istanza di Tomcat. Quindi sarà necessario inserire la libreria di Dynamo e l'archivio *.aar* che consente di effettuare il deploy del VersionConsistencyService con Axis2.