POLITECNICO DI MILANO
Corso di Laurea Specialistica in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Architetture e linguaggi per Runtime Software Adaptation: Computational REST ed Erlang

Relatore: Prof. Carlo Ghezzi
Correlatore: Prof. Gianpaolo Cugola

Tesi di Laurea di:
Alessandro Sivieri, matricola 724745

Anno Accademico 2009-2010

I'm a doctor, Jim, not an engineer!

Sommario

Le recenti evoluzioni delle tecnologie che affiancano ed accompagnano sempre di più le nostre vite ha comportato negli ultimi anni una trasformazione del software, che da prodotto con un funzionamento definito e costante nel tempo è divenuto un prodotto che necessita di adattarsi continuamente al contesto all'interno del quale si trova ad operare. Le applicazioni su larga scala, ed in particolare rivolte ad Internet, necessitano di gestire grandi flussi di dati distribuiti e con utenze elevate, ed al contempo offrire funzionalità tipiche di quelle applicazioni in precedenza costruite per funzionare su singoli computer.

L'Ingegneria del Software deve essere in grado di modificare a sua volta le modalità di definizione, progettazione, sviluppo, test ed evoluzione degli applicativi stessi, in modo da inglobare le nuove necessità che la tecnologia richiede ed al contempo garantire un funzionamento consistente ed affidabile dei propri prodotti.

La presente Tesi si inserisce in questo ambito di ricerca, ed in particolare si concentra sulle architetture in grado di supportare adattamento a runtime del software: una di queste architetture è Computational REST, evoluzione del più noto paradigma REST, che riprende concetti tipici dei sistemi a "codice mobile" per offire funzionalità di adattamento ad applicazioni costruite su larga scala e pensate per Internet.

Questo lavoro analizza le caratteristiche di tale architettura e presenta una implementazione alternativa del framework adatto a supportarla, utilizzando un diverso linguaggio di programmazione, Erlang, un linguaggio funzionale orientato alla concorrenza che rivela alcune interessanti caratteristiche di supporto al software context-aware. Sono quindi presentati i risultati dei test di un caso di studio che utilizza queste due tecnologie assieme e le conclusioni riguardo le capacità del linguaggio e dello stile architetturale analizzati.

Ringraziamenti

Il primo e più importante ringraziamento, inevitabilmente, va ai miei genitori: a loro devo la possibilità che ho avuto di venire qui a Milano a studiare, di frequentare questa Università e di portare a compimento la Laurea Magistrale, ed i miei sentimenti nei loro confronti vanno ben oltre qualunque cosa io possa mai scrivere con questa tastiera in queste poche righe, perché ogni parola, ogni telefonata a tarda sera durante gli esami, ogni rimprovero per una preparazione scarsa da parte mia, ogni complimento per una prova ben riuscita, ogni discussione ha dato il suo contributo a farmi arrivare fin qui. Perciò mi limiterò a sei misere lettere, consapevole del fatto che voi sapete certamente meglio di me cosa c'è dietro a ciascuno di questi simboli: grazie.

Naturalmente, il ringraziamento successivo è per mia sorella, non tanto e non solo per essere stata un sostegno, da "grillo parlante" a compagna di giochi, a seconda della necessità, quanto per augurarle un grosso in bocca al lupo, visto che la prossima ad iniziare l'avventura universitaria sarà proprio lei: che possa essere un periodo proficuo e felice, come è stato il mio; e come tu sei stata presente in questi anni, così lo sarò io, in ogni momento, se e quando ne avrai bisogno.

Un ringraziamento va naturalmente al mio relatore, prof. Ghezzi, per aver seguito il mio lavoro di tesi in questi mesi, per i preziosi insegnamenti e la passione per questo mestiere che i suoi corsi hanno trasmesso, e per la possibilità di proseguire il mio percorso di ricerca anche oltre la Laurea Magistrale, e di cui spero di poter ricambiare la fiducia accordatami; al prof. Cugola, per essere stato mio correlatore e per i prezioni consigli offerti durante le ricerche e lo sviluppo di questa tesi; ai colleghi dell'area di Ingegneria del Software, con cui ho fatto il lavoro di ricerca che ha dato inizio a questa tesi stessa, e con cui ho discusso in questi mesi alcuni aspetti importanti del mio lavoro; al prof. Zanero, per i consigli riguardanti alcuni problemi di sicurezza emersi durante le ricerche e la bibliografia consigliata al riguardo.

Tantissime sono le persone che hanno avuto un ruolo nella mia carriera

universitaria, dalla Laurea ad oggi sono inevitabilmente aumentate e sicuramente mi dimenticherò qualche nome per strada, perciò fatemi precisare subito: voi sapete chi siete, non sarà un nome su un foglio di carta a rendervi più o meno importanti ai miei occhi, in ogni caso chiedo subito scusa se il vostro nome dovesse rimanere impigliato tra i circuiti di questo laptop, e non raggiungere il testo che state leggendo.

Alcune di queste persone hanno, in questo momento, un posto speciale nel mio cuore, e vorrei perciò iniziare ringraziando Nicola, amico prezioso, collega inarrivabile, compagno di suonate, di chiacchierate, di pranzi e cene, di vacanze, di presidenze e Dio solo sa di cos'altro: senza di te, questi sei anni sarebbero stati decisamente più tristi e solitari; Chiara, con cui purtroppo ci vediamo troppo poco negli ultimi tempi, ma che sei sempre stata presente quando ho avuto bisogno di qualcosa, fosse qualcosa di concreto (a tal proposito...) o una semplice chiacchierata; Sante, onnipresente in chat come dal vivo, per uno scambio di vedute, per un consiglio informatico, per un cinema o un aperitivo; Massimo, sistemista di fiducia ma soprattutto compagno di uscite di fiducia; Ale & Miky, che mi hanno iniziato allo sviluppo Web e sono stati impareggiabili colleghi di lavoro; Daniela, con cui ho passato e tutt'ora passerei ore ed ore a chiacchierare tanto di dettagli redazionali quanto del più e del meno, che mi ha presentato le "faccette", questo oggetto misterioso (la cui traduzione italiana è veramente triste...) che ha continuato a riapparire nei miei lavori successivi, ma che soprattutto mi ha mostrato cosa vuole dire fare il proprio lavoro con una passione che spero di poter eguagliare in futuro; Giorgio, che mi ha regalato il mio primo computer, un 286 di cui non posso dimenticare i listati infiniti in BASIC e le giornate trascorse a cercare di "capire": ora ho capito davvero, ma mai 15 anni fa avrei immaginato la fatica che sarebbe stata necessaria!

A fianco di questi amici (e parenti), tante altre persone hanno avuto la loro parte in questi anni: i compagni di appartamento, Marco ed Alberto; i compagni di corso, di feste, di vacanze e quant'altro, Fabio, Ale, Danilo, Beppe, Jack, Desirèe, Alberto, Emanuela; i colleghi del POuL, Radu, Rino, Andrea, Daniele, Federico, Luigi e tutti coloro che hanno contribuito a tutto ciò che abbiamo fatto con l'associazione, dai lan party alle conferenze; gli amici di casa, Matteo, Fulvio, Sara, vari Alberti, Luca e tutti i compagni e le compagne di serata in questi anni, nei weekend passati sul Delta; i colleghi dell'Ufficio Web e di MeglioMilano; i colleghi di KDE Nepomuk, con cui si è discusso e sviluppato un'intera estate e con cui si è convissuto qualche giorno dell'anno scorso (e spero vivamente di trovare un po' di tempo da dedicarci nuovamente); gli amici del dipartimento, Alex, Michele, e tutti quelli che ho conosciuto e di cui non ricordo il nome in questo momento (ma avrò tutto il

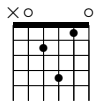
tempo per ricordarmelo, non vi preoccupate!); tutti i parenti a casa, nonni zii e cugini, qualcuno addirittura impegnato in un'avventura universitaria!

Un ringraziamento, infine, va anche alla famiglia Vitucci, per l'ospitalità che mi avete offerto quando siamo stati insieme (a Milano come a Ferrandina), e chissà che non si riesca prima o poi a fare un gemellaggio delle nostre famiglie, evento che non è ancora riuscito!

Chiuderei infine questo lungo elenco con Emiliano, cui va il merito di essere stato la prima persona che ho conosciuto in università, il secondo giorno di lezione del primo anno, e che per questo vorrei ringraziare e nominare per ultimo, per chiudere in qualche modo il cerchio di amici "politecnici".

Non sarò così pazzo da ringraziare i miei computer, che altro non sono che un mero strumento delle mie azioni; nella precedente tesi, avevo ringraziato anche alcuni personaggi del mondo della musica, ma a ben pensarci oggi non posso che ringraziare la Musica in sè, per quanto entità astratta: troppi sono gli artisti di cui ho ascoltato i brani in questi anni, e che hanno accompagnato lo studio o lo sviluppo di progetti, e troppi i compositori di cui ho suonato brani con i miei strumenti, alla ricerca di uno svago tra un esame di meccanica ed uno di statistica, uno di robotica ed uno di logica.

Anche perché, alla fin fine, tutto si riduce a questo: se manca la corrente elettrica, il computer non s'accende, ma la chitarra suona comunque...


Amadd9

Indice

Elenco delle figure

Elenco dei listati

Capitolo 1

Introduzione

Lo sviluppo tecnologico ha portato negli ultimi anni alcuni importanti cambiamenti nel mondo del software: da uno scenario in cui le applicazioni risiedevano nei computer ed elaboravano documenti locali o basi di dati al più a livello aziendale, si è ora passati ad un mondo in cui i programmi risiedono sui dispositivi più disparati, dai cellulari agli smartphone ai laptop ai computer, e questi programmi gestiscono non solo i dati locali, ma sempre più informazioni provenienti da Internet.

Sono applicativi costruiti in modo da lavorare su larga scala, caratterizzati da una connessione persistente ed in grado di gestire flussi di informazioni in ingresso ed in uscita, provenienti da diversi fornitori di servizi; non solo, questi servizi possono variare dinamicamente in base ad esempio al carico del server, o il dispositivo stesso può cambiare fisicamente locazione e trovarsi ad operare in aree in cui il contesto di funzionamento sia differente. Il caso limite è il cosiddetto "Internet of Things", uno scenario in cui la pervasività dei dispositivi è ancora più elevata, e le fonti di dati possono essere anche reti di sensori wireless, o altri apparecchi di uso quotidiano, che oggi magari non immaginiamo neppure essere presenti in una rete.

L'Ingegneria del Software è quella branca dell'ingegneria che segue tutte le fasi di produzione di un programma, dall'analisi dei requisiti alla fase di design, quella di sviluppo, il testing ed il deployment, il mantenimento del software nel tempo. Spesso, essa è costretta a inseguire lo sviluppo tecnologico, e ad adattare le metodologie o a ricercarne di nuove, a seconda di quanto il mercato richieda.

Una pratica soggetta a cambiamenti, ad esempio, è la netta separazione tra la fase di sviluppo e la fase di funzionamento (runtime) del software: tradizionalmente, l'analisi dei requisiti, la modellizzazione e la progettazione

di una soluzione, la verifica formale ed il testing si concentrano nella prima fase, ed una qualunque modifica da applicare alla soluzione rende necessario ritornare dalla fase di funzionamento a quella di sviluppo, così da poter applicare i cambiamenti e verificare nuovamente che le funzionalità siano consistenti con le richieste effettuate.

Oggi, questa divisione netta tra le due parti del funzionamento di un'applicazione sta scomparendo, a causa di diversi fattori: lo sviluppo del software si sta sempre di più decentralizzando, poiché le applicazioni sono spesso composizioni costruite a partire da frammenti sviluppati autonomamente, indipendentemente e da entità separate tra loro, e di conseguenza l'integrazione deve sempre più spesso basarsi su una descrizione più o meno formale del funzionamento dichiarato di quel frammento, con la necessità di verificare se tale descrizione corrisponde a quanto realmente offerto da quel prodotto; al contempo, come abbiamo visto, i dispositivi che collaborano alla fruizione di servizi sono sempre più eterogenei e decentralizzati; i sistemi software che offrono servizi, conseguentemente, devono essere in grado di fornire un livello di informazioni adeguato al tipo di controparte con cui comunicano, per evitare un accumulo di dati ingestibile, ed al contempo il device mobile deve potersi adattare al contesto all'interno del quale si trova ad operare, sia esso una rete di sensori o la rete wireless di un edificio.

Questo cambiamento radicale nella tecnologia e nel funzionamento del software odierno comporta la necessità di cambiamento nelle pratiche ingegneristiche coinvolte nello sviluppo degli applicativi: i diversi momenti della fase di sviluppo, ad esempio la modellizzazione e la verifica formale, devono spostarsi verso la fase di runtime, e nel caso di applicazioni completamente distribuite e decentralizzate, possono essere svolti solamente in quest'ultima fase; non solo, le architetture ed i paradigmi utilizzati nello sviluppo devono cambiare per supportare correttamente l'adattamento a runtime che viene richiesto.

1.1 REST e Computational REST

Una prima fase di ricerca all'interno di questo ambito è stata un'analisi delle esistenti architetture in grado di supportare questo adattamento dinamico: l'articolo [30] ci fornisce un framework in grado di giudicare alcuni paradigmi esistenti oggi secondo alcune dimensioni, come il cambiamento del comportamento dell'applicazione, dello stato di questa, del contesto di esecuzione all'interno del quale essa opera; la mia ricerca si è quindi concentrata su quelli

ritenuti dagli autori i maggiori esponenti della categoria, in particolare REST come stile esistente e Computational REST come stile del futuro.

Quest'ultimo [17], in particolare, è uno stile architetturale che mira a fondere i pregi di REST (REpresentational State Transfer, un insieme di principi che mirano a definire come gli standard Web debbano essere utilizzati affinché sia possibile ottenere la scalabilità del sistema Internet) con i concetti di codice mobile: gli oggetti di scambio tra sistemi sul Web non sono più i contenuti, bensì le computazioni; il singolo host può mostrare all'utente finale un insieme di dati, o può esso stesso eseguire il codice che ha ricevuto e mostrare il risultato di tale esecuzione, o demandare l'esecuzione stessa ad altri peer della rete e ricevere da questi il contenuto finale.

Le problematiche di sicurezza

I paradigmi di codice mobile, e conseguentemente anche CREST, sono inoltre oggetto di interesse per le problematiche relative alla sicurezza informatica: se già allo stato attuale, infatti, il codice che si muove in Internet può creare falle piuttosto serie negli applicativi Web, nonostante tale codice (prevalentemente Javascript) abbia possibilità di interazione con i sistemi degli utenti piuttosto basse, la capacità di spostare intere computazioni aumenta questo livello di interazione ed introduce alcuni potenziali problemi di sicurezza che non possono essere sottovalutati.

L'argomento è stato affrontato in fase di definizione di codice mobile e nei lavori di presentazione di CREST, e varrà ripreso anche per Erlang e per la nuova versione del framework.

1.2 Erlang

Una seconda fase di ricerca è stata un'analisi delle attività di supporto all'adattamento dinamico del software, viste all'interno dei diversi ambiti dell'ingegneria del software, dall'analisi dei requisiti alla fase di testing. Tale ricerca è stata svolta come attività del corso di "Argomenti Avanzati di Ingegneria del Software", ed il campo all'interno del quale, assieme ad altri colleghi, ho svolto tale analisi, è stato quello del supporto offerto dai linguaggi di programmazione al tema di ricerca.

A partire dai risultati evidenziati nel report finale [28], la mia analisi si è quindi concentrata su un linguaggio in particolare, Erlang, che offre diverse caratteristiche utili allo scopo della ricerca stessa, e sulla possibilità di affiancare tale linguaggio con il paradigma architetturale citato in precedenza.

Erlang è un linguaggio di programmazione funzionale ed orientato alla concorrenza [8], il quale offre alcune funzionalità interessanti dal punto di vista di sviluppo di software context-aware, che deve essere in grado perciò di modificare dinamicamente il proprio funzionamento:

- facilità di creazione di processi leggeri, gestiti internamente dal runtime del linguaggio e mappati automaticamente sui core disponibili della macchina che sta materialmente eseguendo il codice;
- facilità di sviluppo di applicazioni distribuite, in particolare il software può essere facilmente portato su un sistema distribuito senza dover cambiare una riga di codice, se le primitive di comunicazione sono state adeguatamente sfruttate;
- capacità del runtime di sostituire moduli "a caldo", ovvero senza necessità di dover interrompere l'erogazione del servizio che quel modulo sta offrendo.

Queste tre caratteristiche, in particolare la terza, sembrano ben sposarsi con le necessità di adattamento dinamico del software al contesto in cui si trova ad operare.

L'interesse si è a questo punto concentrato sul fatto che l'implementazione originale di un framework in grado di operare secondo CREST è stata fatta in Scheme, un linguaggio di programmazione funzionale per il quale gli autori hanno dovuto scrivere diverse librerie, con funzionalità necessarie ad effettuare le interazioni che il paradigma stesso prevede, ma che Erlang offre già come esistenti; l'idea di questa tesi è stata perciò quella di reimplementare il framework utilizzando quest'ultimo, con la possibilità perciò di sfruttare appieno queste funzionalità, e di analizzare quindi pregi e difetti della realizzazione finale, e con essa anche il linguaggio ed il paradigma utilizzati.

1.3 Struttura della tesi

Il secondo capitolo contiene una descrizione dello stato dell'arte riguardo gli elementi che sono stati alla base di questa tesi: il software context-aware, REST e Computational REST, Erlang e le problematiche di sicurezza ad esso legate.

Il terzo capitolo descrive nel dettaglio i principi di REpresentational State Transfer, le problematiche riscontrate nell'applicare tali principi in alcune applicazioni Web, e come Computational REST si propone di superare

questi limiti, introducendo diversi vincoli architetturali e potenziando quelli presenti in REST.

Il quarto capitolo introduce la nuova implementazione di un peer CREST e le motivazioni del passaggio da Scheme ad Erlang come linguaggio di programmazione.

Il quinto capitolo illustra il caso di studio, le problematiche riscontrate durante la realizzazione (e le loro possibili soluzioni), i risultati ottenuti dalle esecuzioni di test del prodotto finale ed un confronto tra le due versioni dei framework in grado di supportare CREST.

Il sesto capitolo discute i pregi ed i difetti di Erlang applicato ad un caso realistico ed i pregi ed i difetti di Computational REST come architettura per le applicazioni Internet del futuro.

Capitolo 2

Stato dell'arte

Come è già stato presentato nell'introduzione, la tesi si muove all'interno dell'area di Ingegneria del Software, in particolare nell'ambito di ricerca di nuovi paradigmi di progettazione e design di soluzioni software che siano in grado di descrivere al meglio le necessità che l'evoluzione tecnologica di questi ultimi anni ha portato.

Lo scenario di riferimento è quello descritto dalle applicazioni Internet su larga scala: esse devono essere in grado di gestire grandi flussi di dati, essere in grado di scalare a fronte di richieste consistenti e conseguentemente variare i servizi offerti sulla base del carico e del tipo di dispositivi interconnessi presenti. I programmi presenti su questi dispositivi devono a loro volta essere in grado di gestire le informazioni che ricevono, informazioni che possono variare a seconda del luogo fisico dove ci si trova, e con quali altri dispositivi si sta interagendo. Lo scenario limite, come già accennato nell'introduzione, presenta un'estrema varietà di device e di capacità computazionali, ed il software deve poter gestire questa varietà e permettere all'utente di usufruire comunque delle informazioni di cui ha bisogno.

Tale software, viene prodotto nell'ambito tecnologico precedentemente descritto, deve essere in grado di adattarsi dinamicamente ai cambiamenti motivati ad esempio dallo spostamento all'interno della rete; una panoramica esaustiva dello stato dell'arte per quanto riguarda i sistemi auto-adattivi e le metodologie che coinvolgono l'Ingegneria del Software può essere trovata in [12].

L'evoluzione del software è una pratica che è sempre esistita, tuttavia in un approccio tradizionale era possibile fermare tale applicazione per effettuare ad esempio un aggiornamento, o in alternativa era possibile costruire sistemi ridondanti, così che fosse possibile fermare solo una parte del sistema, mentre il servizio veniva comunque offerto ininterrottamente agli utenti.

Nel momento in cui, tuttavia, ci collochiamo in scenari altamente dinamici, non è più pensabile poter fermare l'offerta di servizi o costruire sistemi ridondanti, in quanto dispendiosi da un punto di vista di risorse e di sincronizzazione dei dati manipolati. L'evoluzione del software deve avvenire in modo dinamico e soprattutto a runtime, e questa ricerca parte da un'analisi dei meccanismi già esistenti a livello architetturale e di linguaggio di programmazione per affrontare tali cambiamenti, ed ha combinato i risultati più promettenti in entrambi i campi in un'applicazione adatta ad eseguire tali compiti.

2.1 Stili architetturali

Il tema del software in grado di adattarsi a runtime al contesto è stato soggetto ad un'analisi sia alla fine degli anni '90 [29] sia in una ripresa in questi ultimi anni [30], e ci permette di introdurre uno dei due soggetti di questa tesi, ovvero Computational REST.

Uno stile architetturale, in informatica, è un concetto che delinea alcuni elementi significativi all'interno dell'architettura di un software, affrontando alcuni aspetti specifici del sistema hardware o dell'ambito in cui si sta implementando la soluzione richiesta.

L'articolo presenta un framework in grado di comparare diversi stili e modelli architetturali in base anche al supporto hardware (ridondanza), al supporto del linguaggio (capacità dei runtime), ed supporto dei sistemi operativi (virtualizzazione) disponibile; l'articolo si concentra poi sugli stili architetturali in sè, presentandoli come l'approccio più promettente alla progettazione di software altamente adattabile, in particolare grazie alle facilitazioni nell'identificare e rendere manipolabili le parti che possono essere soggette a cambiamenti, al maggior controllo delle interazioni tra parti diverse ed alla gestione dello stato, tutti elementi estremamente importanti nel momento in cui si rende necessario eseguire un cambiamento a runtime.

Gli stili considerati sono diversi:

- *pipe-and-filter*, che coinvolge diversi componenti, i quali instaurano esplicite connessioni tra loro ed elaborano i flussi di informazione in ingresso, inviando i propri risultati al componente successivo; in genere i componenti non mantengono uno stato, quindi ciascuna singola elaborazione è autocontenuta ed i filtri possono essere aggiunti e rimossi in modo arbitrario.

Un'ulteriore evoluzione di questa architettura, chiamata *Weaves*, introduce controlli di flusso e sistemi di buffering per offrire la capacità di ricollegare in modo differente i componenti dinamicamente.

- *publish-subscribe*, un esempio di architettura basata su eventi, in cui non vi è un'invocazione diretta di un componente, quanto piuttosto la notifica di un evento da parte di un componente centrale, il quale si occupa di raccogliere i dati pubblicati e di notificare correttamente i sottoscrittori di tali dati. Questo consente, nuovamente, la possibilità di aggiungere e rimuovere arbitrariamente componenti, dato che la comunicazione non è diretta ed il bus di trasmissione è in grado di gestire dinamicamente i propri utenti.

- *C2*, uno stile in cui ciascun componente è connesso al più con due connettori, uno posto superiormente ed un altro posto inferiormente; ciascun componente può conoscere i componenti posti al di sopra di esso, e conseguentemente può inviare delle richieste specifiche, ma non conosce quelli posti al di sotto, e può solo inviare notifiche a questi ultimi. Questo stile permette di creare software adattabile, grazie alle capacità di creazione di strati ed a regole di visibilità ben chiare.

- *Map-Reduce*, introduce la capacità di parallelizzare computazioni su grandi quantità di dati, distribuendo il lavoro su diversi nodi e collezionando i risultati. Nella fase di *map* ciascun nodo riceve una parte dei dati e restituisce coppie chiave-valore, mentre nella fase di *reduce* il componente centrale è in grado di trovare tutti i valori elaborati in precedenza ed associati ad una specifica chiave, e restituisce il valore finale della specifica elaborazione eseguita.

- *REpresentational State Transfer*, descritto nel dettaglio nel paragrafo successivo.

- *Computational REST*, anch'esso descritto successivamente.

- *Service-Oriented Architectures*, architetture costituite da servizi differenti, organizzati in proprio ed in grado di interagire tra loro utilizzando una rete e protocolli specifici. Ciascun servizio è gestito tramite un'autorità separata, può essere implementato in linguaggi differenti e può essere agganciato e sfruttato a runtime, di conseguenza deve essere in grado di dichiarare le specifiche delle funzionalità offerte.

  Questo introduce alcune necessità, in particolare la fiducia necessaria nello sfruttare un servizio remoto, la possibilità che le specifiche di

quel servizio cambino nel tempo, eventuali problemi di sicurezza legati al servizio stesso; questo ha comportato la nascita di meccanismi in grado di offrire interfacce di comunicazione generiche e la capacità di sfruttare dinamicamente nuovi servizi.

Meccanismi di questo tipo si sono visti in *CORBA*, nei primi anni di SOA, e nei *Web services* oggi, tramite ad esempio i contratti di SOAP, in cui una parte di metadati descrive il servizio, mentre la parte di messaggio è specifica di quell'implementazione e non deve essere trattata dagli intermediari.

- *Peer-to-Peer*, caratterizzato dalla presenza di nodi, chiamati peer, che comunicano tramite una rete privata costruita al di sopra di una rete pubblica, solitamente Internet. In genere, ciascun nodo offre i medesimi servizi ed è indipendente dagli altri, può quindi agganciarsi e sganciarsi dalla rete in modo dinamico; lo stato del sistema è distribuito, e la ridondanza dei dati è utilizzata per ovviare all'assenza di peer che precedentemente erano invece presenti. Diversi meccanismi sono stati sviluppati negli anni per offrire sistemi di ricerca dei dati e coordinazione tra i nodi.

Due sono gli elementi che gli autori hanno tratto dall'analisi degli stili visti finora, e che vedremo effettivamente implementati in CREST: il primo è la necessità che i binding siano variabili, ovvero che ciascuna entità possa essere separata dal proprio contesto fino all'ultimo momento, quando la necessità di computazione rende obbligatoria la nascita di questo legame; non solo, ma tale legame deve poter essere interrotto a fronte di necessità di adattamento; questo quindi favorisce l'adattamento stesso, focus di tutta la ricerca.

Il secondo punto è la necessità che gli eventi ed i messaggi siano elementi primari dell'architettura, che vengano comunicati senza necessità di invocazione diretta di procedure o tramite interfacce specifiche dei singoli servizi, e che possano essere manipolati da intermediari senza che i componenti mittente e destinatario siano in qualche modo coinvolti o risentano di queste manipolazioni; ciascun messaggio/evento può inoltre contenere metadati che permettano l'identificazione dei servizi che devono essere presenti nei riceventi.

2.1.1 REpresentational State Transfer

REpresentational State Transfer (REST [18]) è lo stile architetturale che sottende al World Wide Web, di conseguenza è possibile considerarlo lo stile

maggiormente di successo nel descrivere un contesto di adattamento software su larga scala: il Web è un ambiente estremamente variabile, tanto che non è possibile avere una rappresentazione fedele dell'architettura corrente, questo a causa del numero di dispositivi (client, server, intermedi) che continuamente si aggiungono o vengono rimossi dalla rete stessa.

REST è un'architettura centrata sul *contenuto*: il supporto è orientato allo scambio di contenuti ipermediali all'interno della rete, interconnessi tra loro tramite riferimenti (link); non solo, tale scambio deve avvenire indipendentemente dal numero di componenti intermedi connessi e cercando di minimizzare la latenza; per fare ciò, REST definisce sei principi basilari, qui esposti e che verranno poi ripresi nel dettaglio nel capitolo successivo:

1. l'astrazione chiave dell'informazione è la *risorsa*, specificata univocamente tramite un URL;
2. ciascuna risorsa è rappresentata da una sequenza di byte, con un'ulteriore sequenza di byte che ne fornisce una descrizione;
3. tutte le interazioni sono libere dal contesto;
4. solo poche operazioni basilari sono disponibili;
5. sono incoraggiate operazioni idempotenti e metadati di rappresentazione, per sfruttare meccanismi di caching;
6. è promossa la presenza di intermediari.

L'obiettivo di questi principi è quello di rendere il Web scalabile, a minima latenza ed adattabile a diversi dispositivi. Un sistema che segue questi principi è detto "RESTful".

Negli ultimi anni, tuttavia, alcuni lavori di ricerca hanno iniziato a sottolineare prima di tutto con quale dettaglio i programmi che operano nel Web abbiano integrato al loro interno tali principi [14], e successivamente da un lato quali difficoltà si incontrino nell'adattare tali principi al funzionamento richiesto di alcune applicazioni, e dall'altro quali potenzialità siano state invece rivelate e sfruttate dalle nuove caratteristiche introdotte dal cosiddetto "Web 2.0" [15], dimostrando perciò la necessità che esista un set di principi comuni alle applicazioni che interagiscono in Internet.

2.1.2 Computational REST

Computational REST è un paradigma evoluzione di REST, che sposta l'attenzione dal contenuto alla computazione; prima di descrivere maggiormen-

te nel dettaglio questo stile architetturale, è conveniente introdurre qualche concetto relativo al "codice mobile".

Il "codice mobile" è un concetto ripreso in questi ultimi anni, già oggetto di studio negli anni '90: si tratta di un approccio basato sulla capacità di trasferire codice tra diversi nodi di una rete, rompendo di fatto il legame tra il codice stesso ed il suo ambiente di esecuzione.

Questo argomento è stato analizzato a fondo nella seconda metà degli anni '90, ad esempio in [22], [13], [20]. In particolare, quest'ultimo lavoro descrive un framework in grado di classificare diversi tipi di paradigmi in grado di supportare codice mobile (tra due host, ad esempio), in confronto con il paradigma client-server:

- *client-server*, in cui il secondo host (server) ha il codice, le risorse ed il potere computazionale, ed il primo (client) ha solamente necessità di accedere alle informazioni contenute nel secondo; in questo caso, non si realizza alcun movimento di codice, ma solo di dati (contenuto);
- *remote evaluation*, in cui il primo host ha il codice da eseguire, mentre il secondo host ha le risorse ed il potere computazionale; di conseguenza, il primo invia il codice al secondo, che lo esegue e restituisce il risultato di tale computazione;
- *code on demand*, di fatto il caso opposto al precedente, in cui il primo host non ha il know-how che gli permetta di eseguire il proprio compito, e richiede al secondo host tale conoscenza;
- *mobile agent*, in cui il primo host ha sia il codice che il potere computazionale, mentre il secondo host ha le risorse necessarie per l'esecuzione; di conseguenza, la computazione avviene in quest'ultimo.

Le potenzialità offerte dalle interazioni tra computazioni sono emerse in questi ultimi anni grazie all'utilizzo di AJAX ed alla composizione di servizi (mashup), che secondo gli autori sono due esempi importanti all'interno del quadro delle applicazioni sul Web [17] [16] [23]; Computational REST, perciò, esplora a fondo tale concetto, spostando l'attenzione dal contenuto alla computazione, di fatto riprendendo la modalità di esecuzione remota del codice citata in precedenza: l'obiettivo diviene distribuire e comporre servizi, ed il contenuto stesso diventa una sorta di "effetto collaterale" dell'esecuzione di tali servizi.

L'idea fondante è di mantenere una struttura simile a REST, così da mantenere una compatibilità con questa architettura e con le applicazioni

che oggi operano secondo i principi citati in precedenza, aggiornando tuttavia tali principi secondo la nuova concezione del Web:

1. l'astrazione chiave dell'informazione è la *computazione*, specificata univocamente tramite un URL;

2. ciascuna risorsa è un programma, una chiusura, una continuazione o un ambiente di binding, corredato da metadati che descrivano ciascuna di queste entità;

3. tutte le interazioni sono libere dal contesto;

4. solo poche operazioni basilari sono disponibili, ma è incoraggiato lo sviluppo di nuove operazioni legate alle singole risorse;

5. è promossa la presenza di intermediari.

Il prossimo capitolo descriverà in dettaglio ciascuno di questi punti ed i vincoli strutturali introdotti, soprattutto le differenze e le novità rispetto al paradigma REST di partenza.

2.1.3 Il problema della sicurezza nel codice mobile

Il problema della sicurezza per applicazioni che intendono sfruttare codice mobile è sicuramente rilevante, come già accennato in precedenza nell'introduzione; è possibile trovare una panoramica dei problemi di sicurezza legati al codice mobile in [38], che considera il ruolo di questo all'interno di un ambiente del tipo considerato nella nostra ricerca: due sono gli aspetti da considerare, ovvero la sicurezza dell'host rispetto al codice da eseguire e la sicurezza del codice rispetto all'host; entrambi sono problemi che presentano alcune difficoltà per essere affrontati.

Nel primo caso, è necessario che l'host si difenda da sequenze di istruzioni che possano in qualche modo violarne le proprie policy di sicurezza, tuttavia al contempo l'applicativo potrebbe avere necessità di esecuzione (almeno parzialmente) anonima, per diversi motivi, e questa modalità naturalmente si scontra con la necessità qui espressa; le contromisure note per affrontare il problema sono la firma del codice, l'esecuzione in sandbox e codice in grado di dimostrare la propria aderenza ad un set di policy predefinito.

Il secondo problema riguarda la possibilità che il codice mobile venga manipolato da uno o più host remoti, durante la sua migrazione nella rete; tale codice può essere ispezionato, modificato o usato per portare ad attacchi alle risorse cui ha accesso (database o altre fonti di dati).

Il medesimo articolo offre inoltre una panoramica di ricerche sull'argomento, in particolare riguardo meccanismi di detenzione e prevenzione di attacchi, in un'area di ricerca comunque molto attiva e vivace.

La prossima sezione affronterà una panoramica del linguaggio Erlang, riprendendo in parte questi punti e verificando se tale linguaggio ed il suo runtime offrono un qualche tipo di protezione per l'host o per il codice stesso.

2.2 I linguaggi per realizzare evoluzione dinamica del software

Diversi linguaggi di programmazione e sistemi di runtime offrono costrutti e meccanismi più o meno complessi per effettuare modifiche in modo dinamico: dalla necessità di costruire una panoramica di questi meccanismi è stato sviluppato il lavoro per il corso di "Argomenti avanzati di Ingegneria del Software" [28], che è il punto di partenza per introdurre il secondo soggetto della tesi.

L'articolo prodotto da questo lavoro di ricerca ha classificato alcuni dei linguaggi di programmazione e framework più diffusi secondo tre dimensioni principali:

- il tipo di cambiamenti che il linguaggio permette di effettuare a runtime;
- quali meccanismi possono essere utilizzati per effettuare tali cambimenti;
- quali astrazioni software sono presenti per poter definire tali cambiamenti.

I linguaggi analizzati sono stati suddivisi in tre categorie:

- linguaggi orientati agli oggetti, in particolare *Java* e *C#*, il primo nella versione 7 (non ancora rilasciata) ed il secondo nella versione 4, ed i meccanismi che questi tre linguaggi offrono per effettuare *HotSwap*, quindi modifiche del codice a runtime, oppure l'interazione con il compilatore tramite API, ed infine la possibilità di utilizzare linguaggi dinamici nella medesima macchina virtuale; entrambi i linguaggi hanno dimostrato di offrire funzionalità simili in tutti questi campi, con un leggero vantaggio da parte di Java per quanto riguarda l'aggiornamento dinamico del codice.

- Linguaggi funzionali, in particolare Erlang, le cui caratteristiche verranno introdotte nel dettaglio nel prossimo paragrafo, e Scala, il quale introduce la possibilità di effettuare adattamenti e riutilizzo del software grazie alla possibilità di creare le cosiddette *view*, ovvero metodi aggiunti alle classi che permettono di convertire tipi dinamicamente, dato che tali view vengono inserite dal compilatore in base al contesto in cui si trovano.

- *Aspect-Oriented Programming*, che offre la capacità di separare gli aspetti funzionali di un'applicazione da quelli non funzionali, che tipicamente si collocano in maniera orizzontale rispetto ai primi e difficilmente sono completamente scindibili da questi; l'AOP offre due meccanismi per affrontare questo, ovvero *pointcuts* [27] e *separation of concerns* [21], che permettono di implementare tali funzionalità trasversali in modo veramente separato. Una variante di AOP, chiamata *Dynamic AOP*, permette l'aggiunta e rimozione di queste funzionalità dinamicamente e solamente quando i punti di applicazione sono effettivamente raggiunti.

Erlang è risultato il primo linguaggio riguardo il tipo di modifiche permesse, poiché i meccanismi di hot-swap forniti (spiegati nel dettaglio nel seguito) permettono di fatto la modifica di qualunque aspetto del codice sorgente, e di effettuarla a runtime senza interruzione di servizio, per quanto esso non offra comunque tutte le funzionalità tipiche dei linguaggi Aspect-Oriented.

A partire da queste conclusioni, si è quindi deciso di approfondire Erlang, per verificarne in profondità le funzionalità offerte e l'utilizzo di queste in casi d'uso reali.

2.2.1 I linguaggi funzionali

I linguaggi funzionali sono una famiglia di linguaggi caratterizzati da computazioni effettuate come valutazione di funzioni matematiche, senza uno stato definito ed impedendo la modifica di variabili; nella programmazione funzionale vengono eseguite funzioni i cui risultati, a fronte del medesimo input, sono analoghi, poichè non avviene la modifica di computazioni precedentemente calcolate: non vi sono effetti collaterali, e di conseguenza il comportamento del programma è maggiormente lineare e prevedibile, a differenza dei modelli differenti.

Due sono i costrutti rilevanti che vogliamo definire qui, prima di passare alla trattazione di Erlang:

- le *continuazioni*, sistemi di rappresentazione di uno stato di esecuzione di un programma ad un dato momento: alcuni linguaggi permettono di interrompere l'esecuzione di una computazione ad un dato momento e di riprenderla in un momento successivo; di fatto la continuazione permette al programma di muoversi da uno stato ad un altro, in base alle azioni dell'utente;

- le *chiusure*, ovvero funzioni che contengono alcune variabili libere presenti nell'ambiente in cui queste sono definite: queste variabili restano accessibili per tutta la durata della singola chiusura, persistendo perciò anche a fronte di chiamate multiple alla funzione ivi definita; queste variabili possono perciò essere utilizzate per simulare uno stato o costrutti tipici della programmazione ad oggetti.

Entrambi questi costrutti sono utilizzati per introdurre un concetto di *stato*, altrimenti non presente nei linguaggi funzionali; il primo, come vedremo, sarà l'oggetto centrale della trasmissione di codice di CREST, mentre il secondo verrà utilizzato nell'implementazione del framework in Erlang, poichè questo linguaggio non supporta nativamente le continuazioni.

2.2.2 Erlang

Erlang è un linguaggio di programmazione funzionale orientato alla concorrenza [7] [8], caratterizzato da tipizzazione dinamica, variabili ad assegnamento singolo e valutazione immediata del loro valore.

Da un punto di vista di programmazione, due sono gli aspetti più importanti di questo linguaggio: il primo sono le caratteristiche funzionali e di tipizzazione dinamica, che permettono scrittura di codice estremamente conciso e compatto, caratterizzato dal fatto di essere dichiarativo, ovvero di descrivere *cosa* deve essere computato e non *come*; questo meccanismo viene facilitato anche dal pattern matching, che permette di selezionare tra differenti casi e di estrarre parametri da strutture dati. [11]

Il secondo aspetto importante è la concorrenza: Erlang è in grado di costruire processi leggeri, gestiti internamente dal runtime e mappati dinamicamente su thread di sistema e sui core disponibili in base al tipo di macchina su cui viene eseguito il codice stesso; questo permette di creare una grande quantità di processi differenti in maniera rapida e senza appesantire il runtime stesso, così da parallelizzare il più possibile le applicazioni ed approfittare dell'hardware disponibile in maniera trasparente allo sviluppatore.

I processi leggeri esistenti in Erlang comunicano sfruttando le primitive del linguaggio ed un protocollo proprio della VM che esegue il codice; come

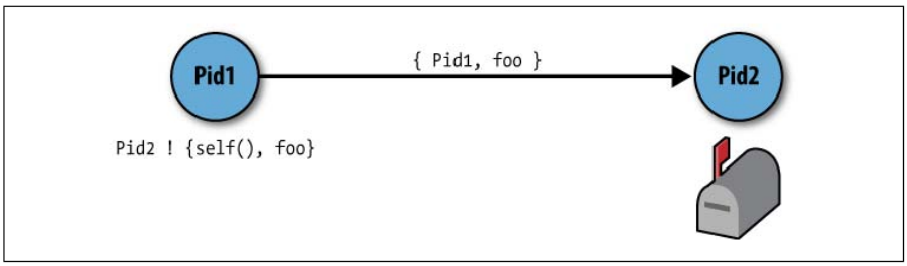


Figura 2.1: Comunicazione nativa di Erlang

mostra la figura 2.1[1], il processo mittente utilizza il costrutto *send* (il punto esclamativo sulla sinistra) per inviare un messaggio alla mailbox del processo ricevente; tale processo può quindi usare la primitiva *receive* per ricevere i messaggi nell'ordine in cui sono stati ricevuti, elaborarli e rispondervi uno per uno.

Come già detto in precedenza, la gestione di variabili con assegnamento singolo permette di non avere memoria condivisa tra più processi, evitando così meccanismi di sovrascrittura di valori o deadlock tra processi stessi; la comunicazione avviene tramite invio asincrono di messaggi, ed il meccanismo di pattern matching permette inoltre di ricevere comunicazioni in maniera selettiva, ignorando ad esempio messaggi che rispondano a determinate strutture particolari.

Un'altra funzionalità interessante è l'affidabilità, dovuta alle caratteristiche del middleware nativo implementato in OTP[2], il quale permette di collegare processi tra loro ed avere sistemi di gestione ad albero, con la possibilità di intercettare eventuali problemi, eccezioni o crash di un processo singolo e di riavviare tali processi senza dover riavviare l'intero applicativo; Erlang è anche in grado di facilitare il porting di un programma in un sistema distribuito: i meccanismi di comunicazione tra processi, infatti, se usati correttamente permettono di sfruttare TCP/IP senza particolari difficoltà, e di avviare quindi i processi leggeri su macchine fisicamente diverse tra loro, comunicanti tramite una connessione di rete secondo un proprio protocollo TCP.

Dal punto di vista di evoluzione dinamica, naturalmente, la caratteristica più rilevante è la capacità del runtime di effettuare hot-swap di codice: nel momento in cui una funzionalità offerta da un modulo viene invocata con il

[1]Illustrazione proveniente da [11].

[2]Open Telecom Platform, la libreria principale sviluppata e distribuita assieme al linguaggio.

nome qualificato (del tipo *module:function()*), il runtime garantisce di eseguire sempre la versione più recente del modulo stesso; se il binario intermedio di tale modulo viene aggiornato durante l'esecuzione, tutte le chiamate alla versione vecchia del modulo rimangono inalterate, mentre qualunque chiamata nuova utilizzerà l'ultima versione del codice. Da notare, comunque, che il runtime non permette la presenza contemporanea di più di due versioni di un medesimo modulo: se ne appare una terza, le istanze della prima vengono terminate, così che la seconda diviene la vecchia versione e la terza quella nuova, come mostrato in figura 2.2[3].

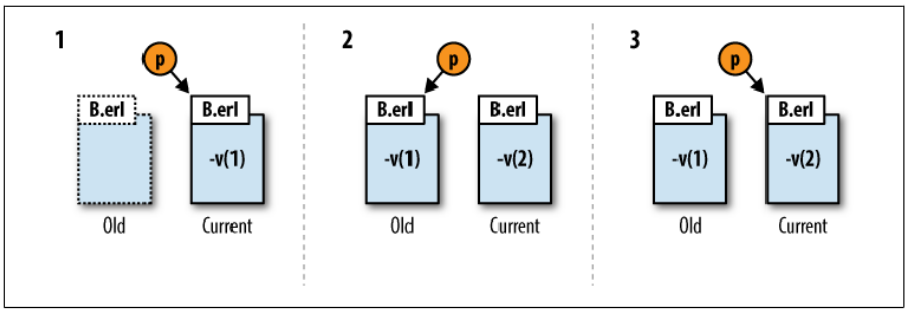


Figura 2.2: Illustrazione di un caso di hot-swap in Erlang

Inoltre, Erlang permette anche il caricamento dinamico di codice che non sia già stato caricato all'avvio dell'applicazione, e vedremo come questa funzionalità sarà utile nella realizzazione del progetto descritto in questa tesi; permette inoltre di gestire un repository di codice, che può trasmettere gli aggiornamenti a tutti i peer Erlang collegati, ed infine permette la creazione di versioni di applicazioni, chiamate release, per le quali è possibile creare aggiornamenti che vengono applicati a runtime, sfruttando di fatto i meccanismi di livello più basso descritti in precedenza per realizzare strutture più complesse e tuttavia automatizzate.

Un'applicazione immediata delle caratteristiche fin qui presentate per Erlang è la realizzazione di server Web: il linguaggio stesso offre alcuni pattern architetturali che, se adeguatamente sfruttati, possono utilizzare le caratteristiche fondanti del linguaggio per creare server Web in cui la sola parte funzionale debba essere sviluppata, demandando le parti non funzionali ed i meccanismi di aggiornamento alla piattaforma.

L'articolo [10] contiene una disamina dei più diffusi framework di creazione di Web server, oltre ad alcuni framework che permettono di costruire siti dinamici seguendo il noto pattern Model-View-Controller (MVC).

[3]Illustrazione proveniente da [11].

La nostra attenzione si è concentrata sulla prima parte, relativa ai server Web, in cui vengono presentati Inets, un server HTTP basilare incluso in OTP, che offre solamente le funzionalità di base necessarie per costruire applicazioni più complicate; Yaws, probabilmente la piattaforma server più famosa scritta in Erlang, che funge da server generico e che supporta anche la creazione di pagine dinamiche; MochiWeb, che supporta JSON e servizi RESTful, e permette di costruire applicazioni personalizzate e leggere, con sistemi di supervisione e di caricamento dinamico delle modifiche effettuate ai moduli.

2.2.3 Il problema della sicurezza in Erlang

Una panoramica di questo linguaggio di programmazione deve necessariamente affrontare il problema della sicurezza: l'implementazione di un paradigma di codice mobile deve infatti, come abbiamo visto in precedenza, affrontare le problematiche che questo comporta, dalla validazione del codice da eseguire, alla trasmissione sicura di questo codice, all'esecuzione in un ambiente controllato.

Erlang, sfortunatamente, non affronta in modo deciso problematiche relative alla sicurezza nè come linguaggio nè come piattaforma di esecuzione: esso nasce infatti come linguaggio adatto a costruire applicazioni che operano in contesti ritenuti sicuri, come ad esempio una rete telefonica aziendale, e di conseguenza non si preoccupa della possibilità che un aggressore possa danneggiare il sistema stesso.

La trasmissione di codice

Il primo problema che insidia Erlang è la trasmissione tra processi: come spiegato in precedenza, la comunicazione nativa può essere portata su un sistema distribuito con estrema facilità ed efficienza, tuttavia il grave problema è che il link dati tra i computer, che viaggia su TCP, non è criptato e di conseguenza può essere soggetto a diversi attacchi piuttosto noti [31]; c'è la presenza di un layer alternativo su SSL, tuttavia una sua implementazione matura è piuttosto recente e non ancora ben integrata nel resto di OTP. Esistono lavori più o meno recenti aventi come obiettivo la produzione di livelli criptati funzionanti, tuttavia in alcuni casi sono lavori piuttosto datati e mai implementati a fondo [32] [34].

In ogni caso, questa prima problematica è marginale rispetto al presente progetto di ricerca, poiché la comunicazione di Computational REST avviene tramite HTTP, di conseguenza è possibile utilizzare un server in grado

di supportare HTTPS per aggiungere un livello criptato al framework, e permettere così al codice di muoversi nella rete senza manomissione.

L'esecuzione del codice

Un problema invece più rilevante riguarda l'esecuzione del codice stesso: la possibilità infatti di eseguire un servizio di provenienza incerta o contenente chiamate a funzioni pericolose per il sistema di esecuzione è ben nota e dovrebbe essere soggetta a controllo.

Erlang non fornisce un ambiente "sandbox", a differenza ad esempio dalla Virtual Machine di Java; alcuni lavori di ricerca si sono perciò concentrati sull'integrazione di meccanismi di sicurezza, tuttavia anche in questo caso si tratta di lavori incompleti o datati.

Il lavoro [9] illustra un'implementazione di un sistema di policy da applicare a nodi Erlang: di fatto, ciascun nodo riceve la richiesta di esecuzione di una determinata funzione, tramite RPC, e decide secondo la propria policy se permettere o meno tale esecuzione; questo approccio richiede l'implementazione di un passaggio ulteriore per richiedere un'esecuzione remota, inoltre sarebbe necessario decidere per un approccio *blacklist* o *whitelist*, infine non si può applicare direttamente al caso di studio, dato che CREST pone in movimento un listato di codice (in particolare una closure), mentre nel caso presentato ciascuna chiamata deve essere a sé stante.

Il lavoro [25] presenta un approccio simile, nuovamente tramite policy e bloccaggio esplicito di chiamate: in questo caso, a livello di codice gli sviluppatori hanno riscritto alcune chiamate di libreria, e dirottano qualunque chiamata alle API standard verso le loro API, di fatto creando un wrapper attorno alle funzioni ritenute pericolose. E' un approccio in parte più adatto al caso di studio, tuttavia il progetto è abbandonato da diverso tempo e ripropone la necessità di implementare tutte le funzionalità che si vogliono permettere.

Infine, [24] propone un'implementazione di Access Control List in Erlang, la quale tuttavia richiede la reimplementazione di parte del runtime stesso di Erlang; di nuovo, una realizzazione di quanto discusso non è disponibile o non è comunque stata integrata nella distribuzione standard di Erlang.

Capitolo 3

REST e Computational REST

3.1 REpresentational State Transfer

Il World Wide Web, così come lo conosciamo oggi, basa la propria struttura su uno stile architetturale chiamato REST, REpresentational State Transfer, che guida il comportamento di tutti gli agenti che agiscono in questa Rete, siano essi applicativi per fornire servizi o per sfruttare servizi.

Il focus di REST è il *contenuto*, poiché si trova ad operare in un mondo di dati interconnessi tra loro, e suo compito è rendere questi dati fruibili a chiunque ne faccia richiesta, minimizzando la latenza e massimizzando lo sfruttamento delle risorse a disposizione. Per ottenere la scalabilità che Internet richiede, questo stile architetturale è stato progettato come fusione di diversi stili più semplici, ed è probabilmente questo il "segreto" del suo successo.

3.1.1 I principi architetturali

REST è stato progettato a partire da una visione di insieme del Web, cui sono stati aggiunti successivamente alcuni vincoli che hanno portato a definire i principi sotto elencati [18].

Il primo vincolo è la struttura client-server, che caratterizza la connessione tra agenti: questo comporta la separazione netta tra l'interfaccia utente ed i dati a disposizione, e questa indipendenza permette ad entrambe le parti di evolvere separatamente, rendendo ad esempio maggiormente portabile l'interfaccia da un lato, e scalabile l'ambiente server dall'altro.

Il secondo vincolo è il rendere la comunicazione stateless: ciascun messaggio che muove dal client al server deve contenere tutte le informazioni di stato necessarie ad interpretare la richiesta, di fatto trasferendo al client l'onere di mantenere uno stato (ove richiesto). Questo permette di raggiun-

gere una maggiore scalabilità, dato che il server può liberare risorse dopo ciascuna chiamata, una maggiore affidabilità, poiché si semplifica il compito di recupero dello stato corrente se avviene un problema al server stesso, ed una semplificazione anche nel caso sia necessario analizzare il traffico, dato che ciascuna comunicazione è a sè stante.

Il terzo vincolo è legato ai meccanismi di caching: un sistema dotato di cache può infatti sfruttare meglio le risorse di rete, liberando al contempo il server di parte del lavoro di elaborazione dati e dando al client un'impressione di maggiore performance; questo implica che ciascuna comunicazione in risposta ad una chiamata debba essere marcata come *cacheable* o meno, e nel primo caso essa può essere salvata e riutilizzata a fronte di una richiesta analoga.

Il quarto vincolo è un'interfaccia uniforme: l'idea è di fornire un'interfaccia generica ed uniforme tra tutti i componenti che agiscono nel Web, così da migliorare la visibilità delle interazioni e l'evoluzione autonoma di ciascuno di questi componenti; al contempo, tuttavia, questo impedisce in parte l'efficienza, data la necessità di utilizzare dati codificati secondo un modello standard; questo vincolo rende il WWW estremamente efficiente riguardo il movimento di contenuti, ma non altrettanto efficiente a fronte di un diverso tipo di interazioni, ed è proprio in questa problematica che Computational REST, come vedremo, si inserirà e proporrà una soluzione.

Il quinto vincolo richiede la possibilità di costruire sistemi a strati: la comunicazione tra server e client deve poter avvenire con la presenza di intermediari, e tale presenza deve essere nascosta il più possibile ad entrambi; questo permette a ciascuno strato di non preoccuparsi degli altri strati con cui potrebbe trovarsi in comunicazione, di evolvere perciò autonomamente, ad esempio favorendo il mascheramento e l'interazione trasparente tra servizi nuovi e sistemi legacy presenti nella Rete. Questo permette perciò agli intermediari di modificare attivamente i contenuti dei messaggi che ricevono, ad esempio modificandone la codifica per supportare l'interazione tra agenti inizialmente non compatibili tra loro.

L'ultimo vincolo riguarda infine il code-on-demand: è permesso ai client di estendere le proprie capacità tramite l'esecuzione di codice client-side, come script ed applet; questo permette ad esempio lo sviluppo di client generici, le cui funzioni possono essere specializzate in un secondo momento. Tuttavia, è da notare come questo vincolo sia opzionale, poiché riduce la visibilità dei client e potrebbe impedire la corretta fruizione di un contenuto, ove il client non sia in grado di interpretare correttamente il codice che riceve.

Questa serie di vincoli ha prodotto la definizione dei seguenti princi-

pi riassuntivi, che abbiamo già proposto precedentemente e che ora hanno maggior senso, definite le necessità appena presentate:

1. l'astrazione chiave dell'informazione è la *risorsa*, specificata univocamente tramite un URL;

2. ciascuna risorsa è rappresentata da una sequenza di byte, con un'ulteriore sequenza di byte che ne fornisce una descrizione;

3. tutte le interazioni sono libere dal contesto;

4. solo poche operazioni basilari sono disponibili;

5. sono incoraggiate operazioni idempotenti e metadati di rappresentazione, per sfruttare meccanismi di caching;

6. è promossa la presenza di intermediari.

3.1.2 Le problematiche di REST

Non tutte le applicazioni che interagiscono con il Web sono tuttavia state in grado di recepire a fondo quanto espresso dallo stile architetturale finora presentato, ed alcune di queste richiedono ad esempio interazioni stateful o mancano di supporto a sistemi di caching.

Gli autori di Computational REST hanno rilevato in [16] alcune criticità nello sviluppo di applicazioni particolari:

- il primo caso riguarda la creazione di un modulo di Apache per la gestione di un archivio di mailing list, con necessità di scalabilità piuttosto elevata a causa del numero consistente di accessi; una prima fase di analisi ha portato a verificare come i prodotti esistenti non fossero RESTful, non rispettando alcuni dei principi base, e la fase di sviluppo successivo ha mostrato come sia stato necessario introdurre altri due vincoli aggiuntivi, ovvero una rappresentazione dinamica dei messaggi originali, che permetta la generazione di differenti rappresentazioni per ciascun messaggio, con diversa complessità a seconda di quanto richiesto dal client, e la definizione di un namespace consistente, così da evitare che un riordinamento dell'indice dei metadati possa portare ad inconsistenze.

- Il secondo caso riguarda l'implementazione del protocollo WebDAV per server in grado di gestire il sistema di controllo di versione Subversion:

tale implementazione non seguiva correttamente i principi REST, poichè la libreria utilizzata per la comunicazione HTTP non aveva supporto completo per il *pipelining*, previsto invece dal protocollo, e gli autori avevano perciò dovuto introdurre un nuovo metodo, il quale tuttavia incapsulava dati e metadati in XML, rendendo difficile effettuare operazioni di caching ed introducendo latenze sia sulla rete che sui sistemi client e server, che avevano necessità di costruire e smontare le informazioni inviate.

Anche i Web services, modalità di offerta di servizi che ha avuto un incremento notevole negli ultimi anni, spesso violano i principi REST: se infatti questi operano utilizzando SOAP come protocollo di scambio messaggi, spesso lo impiegano assieme a HTTP POST, che è un'operazione non idempotente e di conseguenza non cacheable; inoltre, tale protocollo richiede che parte dei metadati relativi al contenuto del messaggio siano codificati in alcuni particolari campi dell'entità XML che costituisce il messaggio SOAP, violando la separazione tra dati e metadati richiesta da REST ed obbligando eventuali intermediari all'apertura del contenuto per poter visionare queste informazioni. Anche qualora il servizio venga offerto in modalità RESTful, in genere non rende disponibili tutte le operazioni che invece sono invocabili tramite SOAP, di fatto costringendo lo sviluppatore a rivolgersi nuovamente a SOAP.

D'altro canto, vi sono anche alcune applicazioni, emerse con il cosiddetto "Web 2.0", che hanno dimostrato le capacità di REST e ne hanno sfruttato appieno i meccanismi proposti: AJAX, ad esempio, ha implementato efficacemente il paradigma del codice su richiesta, estendendolo e di fatto spostando parte della computazione dal server al client, in maniera sicuramente più estesa rispetto al passato e dando vita ad interazioni lato utente impensabili qualche anno fa; i mashup hanno a loro volta modificato la definizione di intermediario, che ora assume un ruolo maggiormente attivo nel momento in cui redireziona parte della propria computazione per trasferire risultati di servizi da altri host verso il chiamante.

Proprio da queste osservazioni, gli autori si sono mossi per estendere REST ed adattarlo alle evoluzioni che il Web ha presentato negli ultimi anni, spostando l'accento sulle potenzialità del codice mobile integrato su larga scala ed, infine, su Internet.

3.2 Computational REST

Computational REST sposta il focus dal contenuto alla *computazione* [16]: il Web viene costruito, secondo questo paradigma, come interazione e composizione di servizi, ed i contenuti diventano una sorta di effetto collaterale delle computazioni stesse.

Il primo passo in questa direzione è l'impiego di forme di codice mobile più potenti rispetto a quanto si utilizza, ad esempio, in AJAX: lo scambio di computazioni avviene tramite lo scambio di *continuazioni*; questo tipo di meccanismo è implementato in diversi linguaggi, tra cui Scheme [35], che è il linguaggio del framework di riferimento in grado di supportare questo stile architetturale.

Poiché CREST è un'architettura, essa non costringe l'utente all'uso di un singolo linguaggio, ed i meccanismi di creazione di continuazioni possono essere sostituiti dall'impiego di *chiusure*.

3.2.1 Principi

Per facilitare l'implementazione di servizi in grado di operare secondo il nuovo paradigma, vengono anche qui definiti alcuni principi, che riprendono quelli visti in precedenza per REST e ne spostano l'accento verso le computazioni:

1. L'astrazione chiave dell'informazione è la *computazione*, specificata univocamente tramite un URL

Questo principio non è completamente distinto da quanto già esiste in REST, specie pensando ai Web Services, che sottendono computazioni; tuttavia qui si esplicita il passaggio già citato dal contenuto alla computazione.

Quest'ultima è svolta secondo due modalità: *remote* indica la volontà di eseguire la chiusura ricevuta con i parametri associati alla chiamata stessa, e di ritornare al chiamante il valore di tale elaborazione; *spawn* invece richiede l'associazione nel peer ricevente di un processo in grado di eseguire tale chiusura con un URL univoco; qualunque chiamata successiva a tale URL porterà all'esecuzione della chiusura stessa, secondo gli eventuali parametri specificati, e ritornerà al chiamante il risultato dell'elaborazione. Di fatto, questa seconda modalità genera una nuova operazione disponibile sul peer, che può essere eseguita a richiesta.

Il processo che gestisce una computazione generata tramite *spawn* gestisce le proprie richieste secondo l'ordine in cui arrivano: esso è infatti dotato di una mailbox che utilizza per ricevere le richieste, elaborarle e rispondere a ciascuna di esse.

2. Ciascuna risorsa è un programma, una chiusura, una continuazione o un ambiente di binding, corredato da metadati che descrivano ciascuna di queste entità

Ciascuna computazione può essere accompagnata da parametri o ambienti di binding, in grado di associare nomi a valori o a funzioni, così da guidare l'esecuzione della computazione stessa per ottenere il risultato richiesto.

3. Tutte le interazioni sono libere dal contesto

Questo requisito è analogo a quello di REST: sta al messaggio portare con sè tutte le informazioni necessarie per mantenere un eventuale stato; non solo, CREST da questo punto di vista compie un passo ulteriore in avanti, poiché lo scambio di continuazioni può permettere l'interruzione di una computazione e la sua ripresa in un secondo momento, e lo spostamento di continuazioni comprende anche lo stato di esecuzione stesso.

4. Solo poche operazioni basilari sono disponibili, ma è incoraggiato lo sviluppo di nuove operazioni legate alle singole risorse

REST basa il proprio funzionamento sulle due operazioni standard presentate in precedenza, tuttavia CREST permette l'installazione di nuove operazioni su ciascun peer, operazioni che successivamente rispondono alle richieste esterne o possono essere a loro volta utilizzate all'interno di computazioni; inoltre, anche l'utilizzo di nuovi protocolli viene semplificato: se un peer necessita di una elaborazione da parte di un secondo peer, e quest'ultimo non tratta un certo tipo di dato o non supporta un certo tipo di protocollo, il primo può inviare al secondo le routine in grado di interpretare tali dati, così da avere un terreno comune su cui collaborare come computazioni.

5. E' promossa la presenza di intermediari

Anche questo punto ha caratteristiche simili al corrispettivo di REST, con in più l'accento spostato verso la composizione di computazioni.

Conseguenze nel design di applicazioni

L'introduzione dei principi sopra descritti comporta alcune conseguenze interessanti da analizzare per l'implementazione di applicazioni che agiscano secondo questo paradigma.

Nomi in CREST: come già descritto in precedenza, le computazioni sono specificate nuovamente a partire dagli URL; la soluzione che permette di restare compatibili con l'attuale struttura di Internet è includere la computazione negli URL stessi, così da permettere agli attuali client Web di

interpretare tali indirizzi ed eventualmente redirezionarli verso gli interpreti CREST corretti. Tali URL, naturalmente, sono consumati direttamente dalla macchina e non sono adatti all'interpretazione umana; nomi maggiormente intuitivi dei servizi devono essere forniti da strati successivi delle applicazioni.

Un medesimo servizio può essere offerto da un peer tramite diversi URL, ciascuno in grado di offrire ad esempio parametrizzazioni diverse, formati di interscambio diversi o addirittura capacità di monitoring (se l'operazione richiesta è molto lunga) o di debugging per gli autori del servizio stesso.

Un servizio può variare nel tempo: il livello di precisione, se si tratta ad esempio di un algoritmo matematico, può cambiare ed adattarsi ai livelli di carico del server (da cui dipende naturalmente la capacità computazionale a sua disposizione), se si tratta di un servizio dipendente da dati casuali o che cambiano nel tempo, la sua risposta a fronte dei medesimi parametri non potrà nuovamente essere la stessa; al contempo, un servizio può mantenere uno stato tra le sue diverse chiamate, ad esempio se si tratta di un servizio di conteggio di qualcosa.

I servizi devono poter garantire l'indipendenza e la non corruzione dei dati in chiamate parallele, eventualmente utilizzando meccanismi di sincronizzazione tra le chiamate stesse (forti o deboli, a seconda che il sistema gestisca più richieste di lettura rispetto a quelle di scrittura o meno).

La composizione di continuazioni avviene tramite URL, esattamente come nei contenuti intercollegati del Web: ciascuna computazione può fare riferimento ad altre ed invocarne i servizi se necessario.

Gli intermediari, esattamente come in REST, devono essere trasparenti rispetto ai peer: un intermediario può infatti effettuare operazioni di caching o di ispezione del servizio stesso, aggiungere o togliere routine di debug per analizzare o velocizzare l'esecuzione di un servizio, ed i peer ai lati della comunicazione non devono essere influenzati da queste operazioni intermedie.

I peer CREST sono incoraggiati, infine, ad implementare meccanismi di migrazione di computazioni, ove si ritenga necessario per diminuire la latenza o aumentare lo scaling dell'applicazione stesssa, ed al contempo devono garantire meccanismi per verificare la distanza temporale trascorsa tra l'interruzione di una computazione e la ripresa della sua esecuzione, eventualmente specificando una sorta di "scadenza", oltre la quale detta computazione non può più essere ripresa.

Da notare, inoltre, che i due punti citati nel capitolo 2, ovvero la necessità di binding dinamici ed i messaggi come elementi caratterizzanti un'architettura in grado di adattarsi al contesto sono entrambi rispettati da CREST: la possibilità, infatti, di trasmettere una continuazione, quindi una computazio-

ne con codice e stato di esecuzione, e la capacità di interrompere un'elaborazione, realizzano la richiesta di binding dinamico, mentre i messaggi restano incapsulati in HTTP e contengono metadati in grado di garantire loro un'elaborazione da parte di intermediari, come peraltro richiesto esplicitamente dal quinto principio riportato nel paragrafo precedente.

3.2.2 Ulteriori specifiche

Per concludere questa panoramica di Computational REST, due ulteriori aspetti devono essere analizzati.

Il primo riguarda i peer "forti" (*exemplary)* ed i peer "deboli" (*weak*), concetto presentato nella demo di [16]: i primi sono di fatto gli host con un'implementazione completa del framework CREST, e sono perciò in grado di gestire le due operazioni presentate in precedenza e di effettuare il movimento vero e proprio di computazione; i secondi invece sono comuni client Web, ad esempio browser, che perciò si aspettano di ricevere pagine HTML e codice Javascript, ovvero i contenuti che sono in grado di interpretare. I peer forti devono perciò essere in grado di fornire quanto richiesto, a seconda del tipo di host con cui sono in comunicazione.

Il secondo aspetto riguarda la sicurezza: come abbiamo visto nel capitolo precedente, esiste una letteratura in continua evoluzione riguardo le problematiche di sicurezza da applicare al codice mobile, in particolare la protezione dell'host rispetto al codice e viceversa; su questo aspetto, tuttavia, gli autori non forniscono molti particolari, salvo sottolineare come siano vitali l'impiego di sistemi di autenticazione e di restrizione di risorse, e riguardo quest'ultimo punto sottolineano come l'impiego di un interprete Scheme che si poggia sulla Virtual Machine di Java permetta loro di utilizzare i meccanismi di sandboxing già presenti. Ulteriori meccanismi di code inspection potrebbero essere necessari per realizzare CREST, tuttavia dettagli sulla loro eventuale implementazione non sono presenti.

3.2.3 Le recenti evoluzioni: *fractalware*

L'articolo [23], più recente rispetto ai precedenti, presenta alcuni dettagli tecnici ulteriori riguardanti Computational REST, alcuni dei quali vale la pena analizzare più nel dettaglio.

La principale caratteristica che viene descritta qui è l'architettura *fractalware*, ovvero la strutturazione delle interazioni tra un peer ed i processi che egli avvia per eseguire le continuazioni (operazione di *spawn*) secondo una modalità ad albero simile ai frattali.

Come mostrato in figura 3.1[1], ciascun processo avviato viene chiamato *subpeer*, e di fatto incorpora tutte le funzionalità del peer padre, ovvero le capacità di avviare a sua volta processi, di comunicare con altri peer e di rispondere a richieste riguardo la propria computazione. Nella figura, i processi P, V e W sono peer padri, mentre i riquadri interni sono i subpeer (fino a tre livelli all'interno di P).

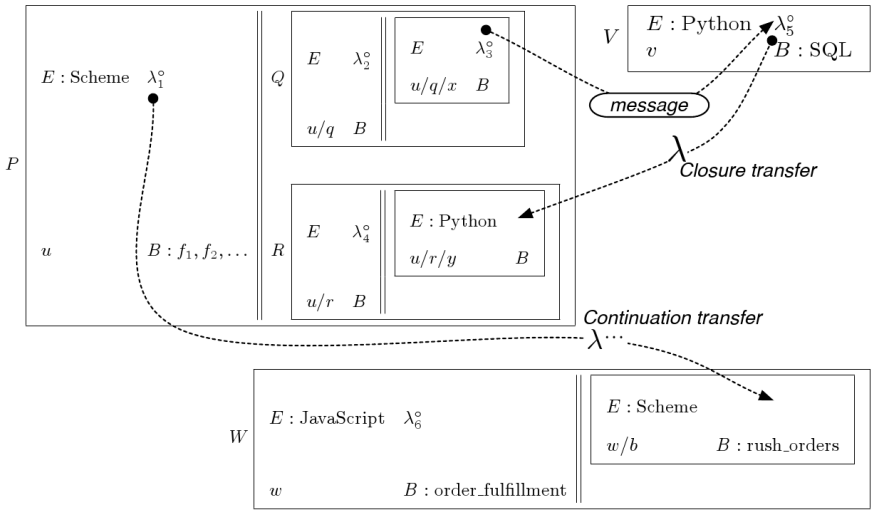


Figura 3.1: Interazione fra peer e subpeer

Il subpeer è in parte limitato dalla propria posizione nell'albero dei processi: non può infatti avere policy di sicurezza o capacità superiori al proprio processo padre, collocandosi quindi all'interno di una gerarchia di capacità.

Un (sub)peer, inoltre, potrebbe eseguire computazioni su una macchina virtuale specifica per un certo linguaggio, permettendo quindi a CREST di essere agnostico rispetto al linguaggio di programmazione utilizzato, così come deve essere uno stile architetturale ispirato a REST.

Un secondo aspetto interessante è la sicurezza degli host che eseguono codice: viene innanzitutto approfondito il ruolo dell'ambiente di binding, definendo anche un ambiente globale proprio di ciascun (sub)peer, che definisce limitazioni nell'ambito del tipo di comunicazioni che esso può effettuare verso l'esterno, nelle funzioni che esso può invocare ed in generale in qualunque tipo di limitazione che sia possibile definire; in alcuni casi, questo ambiente di binding può contenere una sorta di capability list, che descrive appunto le capacità che quel nodo ha nel suo ruolo di peer CREST.

[1]Illustrazione proveniente da [23].

Vengono anche riprese le problematiche sulla sicurezza e, seppur senza dettagli di come le soluzioni ricercate vengano implementate, tre sono i punti principali:

- sandbox: la JVM su cui è eseguito l'interprete Scheme offre nativamente un ambiente sandbox, che può essere utilizzato per impedire danneggiamenti del sistema da parte del codice;
- verifica del bytecode che viene effettivamente inviato al peer;
- self-certifying URLs [26] possono essere impiegati per mutua autenticazione tra peer e subpeer.

Nel prossimo capitolo affronteremo le motivazioni di una reimplementazione del framework Computational REST da Scheme ad Erlang, tenendo comunque presente che l'implementazione di riferimento fornita dagli autori è parziale, di conseguenza alcune decisioni sono state prese autonomamente per costruire un'applicazione di esempio funzionante ed al contempo in grado di seguire correttamente i principi sopra esposti.

Capitolo 4

CREST: da Scheme ad Erlang

Terminata l'analisi delle caratteristiche salienti dell'architettura Computational REST, in questo capitolo si discutono le scelte implementative effettuate dagli autori del framework originario [17], confrontando il linguaggio di programmazione e le librerie da loro scelte con il linguaggio che questo lavoro propone come alternativo, per implementare in modo più efficiente un framework che segua le linee guida di CREST.

4.1 L'implementazione di riferimento

Computational REST utilizza Scheme, un linguaggio di programmazione funzionale dialetto di Lisp, come linguaggio principale per lo sviluppo di un framework che supporti questa architettura.

La scelta è apparsa piuttosto naturale, dato che diversi lavori precedenti si sono occupati di scambio di computazioni e si sono appoggiati a tale linguaggio, in particolare grazie al supporto nativo per le continuazioni in esso presente [35]. La chiamata *call/cc*, infatti, permette ad un programma Scheme di riprendere o eseguire nuovamente un set di istruzioni, mantenendo al contempo lo stato di esecuzione presente nel momento in cui tale computazione era stata interrotta.

Come detto, diversi progetti sviluppati precedentemente (Tubes [6], MAST [37], Dreme [19]) hanno sfruttato questo meccanismo per spostare computazioni tra host all'interno di reti, realizzando appieno il paradigma di codice mobile e costituendo perciò una base di partenza per l'implementazione stessa di CREST.

Una seconda caratteristica interessante nell'utilizzare Scheme è stata la possibilità di sfruttare un interprete del linguaggio implementato per la Java Virtual Machine: questo ha permesso di sfruttare il sandboxing nativo fornito

da tale VM ed un ulteriore secondo livello di protezione fornito da Scheme stesso nei confronti di Java.

Il fatto, inoltre, di avere un interprete Java ha permesso di sfruttare alcune librerie già esistenti per quest'ultimo linguaggio, ad esempio alcuni parser di feed (RSS, Atom) o librerie di comunicazione I/O (*New I/O*, JSR 51).

Gli autori hanno inoltre sviluppato un'applicazione demo, un feed reader che può essere condiviso e reso collaborativo, sfruttando le capacità dei peer CREST che lo sostengono.

Tale demo presenta la suddivisione tra peer forti e peer deboli, dove i primi mantengono uno stato mentre i secondi sono utilizzati nell'interfaccia grafica; ciascun client di fatto condivide la medesima computazione, modificandone lo stato tramite l'interfaccia ed i peer deboli, che successivamente sincronizzano tutte le modifiche verso gli altri peer. In ogni momento, ciascun peer forte può essere duplicato (*deep copy*) e la nuova istanza esegue una nuova computazione, copia esatta e distinta rispetto a quella di partenza, mantenendo perciò più istanze e più stati diversi.

In questa demo, l'interfaccia grafica è sviluppata tramite AJAX, e l'utilizzo di Javascript dimostra come sia possibile utilizzare linguaggi differenti da Scheme per ottenere comportamenti compatibili con CREST, in questo caso utilizzando chiusure anziché continuazioni.

Durante l'analisi di Computational REST, ci siamo chiesti se Scheme risponda veramente alle esigenze di adattabilità del codice che il framework stesso si troverà a sostenere, o se esistesse un linguaggio avente potenzialità simili o addirittura superiori per lo sviluppo di applicazioni in codice mobile.

Scheme è un linguaggio funzionale caratterizzato dall'avere uno stile minimalista, ovvero avente un nucleo standard piuttosto ristretto, che ne descrive le principali funzionalità, e lascia la gran parte delle caratteristiche ad estensioni del linguaggio stesso.

Questo ha quindi reso necessario agli sviluppatori di CREST di dover definire tutti i moduli necessari a supportare la loro architettura: dai parser JSON (necessari per intercomunicare con Javascript) a quelli dei feed, dalla definizione stessa di *peer* a quella di *fiber*, ovvero un thread leggero in grado di eseguire una certa computazione, alla definizione della *mailbox* che riceve i messaggi inviati a ciascun processo.

Questo ha comportato la creazione di una notevole quantità di codice, nonostante Scheme sia un linguaggio funzionale e di conseguenza la notazione sia in genere più compatta rispetto ai linguaggi imperativi o ad oggetti;

l'implementazione della demo, come citato negli articoli di presentazione della stessa, ha richiesto di per sè poche righe di codice, tuttavia il prodotto finale è comunque notevole da questo punto di vista.

4.2 Il passaggio ad Erlang

Come già citato in precedenza, l'analisi dei linguaggi di programmazione [28] aveva prodotto un notevole interesse verso Erlang, di cui sono già state citate le potenzialità che offre nel progettare e sviluppare applicazioni adattabili al contesto ed in grado di modificare il proprio comportamento a runtime.

Un'implementazione di CREST in Erlang può sfruttare alcune delle strutture native che Erlang ed OTP offrono, evitando perciò di dover definire nel dettaglio tutti i moduli che effettivamente sono necessari al framework; in particolare:

- Erlang contiene nativamente il concetto di *processo leggero* (*fiber* nel linguaggio di CREST), e questi processi sono implementati secondo modalità estremamente efficienti e la loro mappatura dinamica sui core presenti a livello hardware permette una rapida adattabilità alla piattaforma su cui si esegue il peer stesso;
- ciascun processo leggero ha associata una mailbox, la quale supporta la ricezione in coda di messaggi secondo il protocollo nativo di Erlang stesso (un canale TCP sulla porta 4369);
- le librerie OTP permettono il caricamento dinamico di codice e la trasmissione di chiusure, sfruttando i costrutti *lambda* del linguaggio, rendendo perciò realizzabile il movimento di codice tra peer;
- sia OTP che terze parti hanno sviluppato alcuni framework per il Web [10], che supportano la trasmissione su HTTP e la gestione distribuita delle richieste su processi separati.

A questo, naturalmente si aggiungono le capacità di aggiornamento dinamico del codice, di cui si è parlato nel capitolo 2 e che permettono un ulteriore passo avanti nella gestione del framework CREST nel suo complesso; tali meccanismi, inoltre, permettono l'aggiornamento dei moduli che realizzano le computazioni espresse tramite l'operazione di *spawn*.

Queste caratteristiche, unite alla volontà di testare il linguaggio e le sue librerie in un progetto realistico, hanno portato alla scelta di creare un nuovo framework che rispettasse da un lato i principi ed i vincoli strutturali di

Computational REST, e dall'altro fosse in grado di sfruttare, ove possibile, i costrutti offerti ed i moduli nativi già esistenti.

4.3 Progetto del nuovo framework

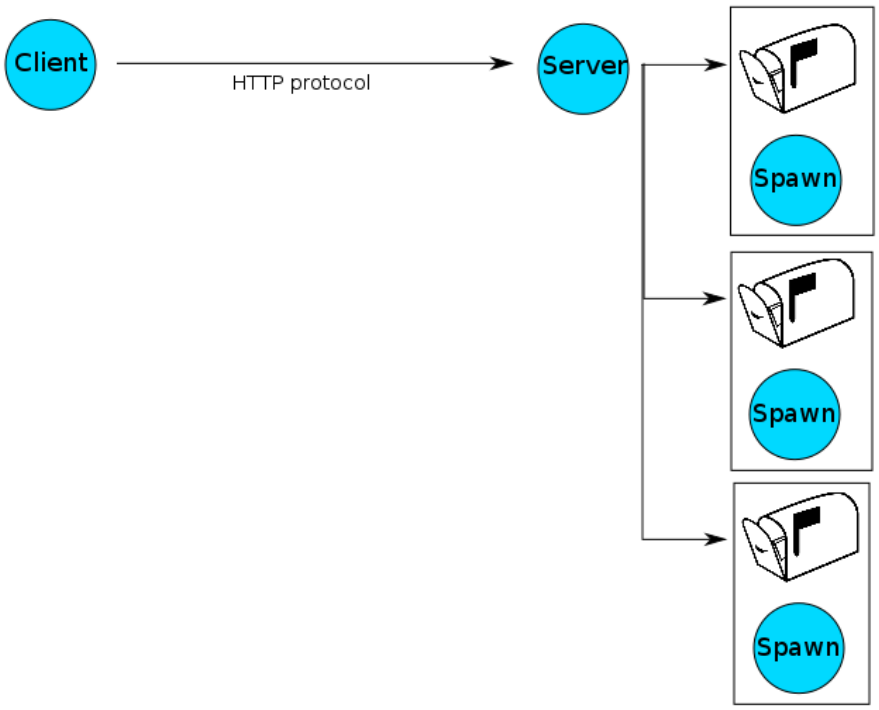


Figura 4.1: Comunicazione tramite protocollo HTTP

La figura 2.1 faceva riferimento a come avvenga la comunicazione in Erlang; la figura 4.1 mostra come la comunicazione avvenga invece nel framework CREST sviluppato in Erlang: i client contattano il Web server utilizzando il protocollo HTTP ed i suoi metodi standard, ed il server successivamente smista le richieste ai processi che sostengono i servizi lanciati tramite *spawn* e *remote*, sfruttando la comunicazione nativa e di conseguenza mailbox e processi leggeri propri del linguaggio. Il framework, perciò, esegue una sorta di incapsulamento della comunicazione nativa su HTTP, permettendo di sfruttare le librerie OTP e di rispettare le richieste di Computational REST.

In aggiunta alle librerie OTP, distribuite nativamente con la piattaforma Erlang, il progetto utilizza due librerie esterne:

- la più importante è *MochiWeb* [4], una libreria per gestire server Web: lo scheletro del framework utilizza questo codice per automatizzare la gestione dei processi in fase di ricezione delle richieste e costruire un'applicazione secondo i *behaviour* di Erlang, permettendo ad esempio la gestione automatica degli aggiornamenti o delle dipendenze;
- *log4erl* [2], un progetto che permette di gestire log interni tramite file e livelli di avviso, in modo simile ad *Apache log4j*.

Di queste tre librerie, l'unica essenziale è la prima, mentre la seconda è stata utilizzata per comodità e la terza per evitare di dover riscrivere codice già esistente; MochiWeb è (ad oggi) un progetto attivamente sviluppato ed alla base di alcuni servizi online di advertisting e distribuzione Flash [3]; si è optato per questa libreria anziché *Yaws*, il Web server standard di OTP, data la sua leggerezza e la possibilità di creare uno scheletro di applicazione Web rapidamente e con una separazione del prodotto in moduli piuttosto chiara.

L'unico difetto principale è la mancanza di supporto a pagine dinamiche, a differenza ad esempio di Yaws stesso, e questo ne limita in parte le capacità per la creazione di parti Web compatibili con REST; in ogni caso, questa mancanza non ha alcuna influenza ai fini dell'implementazione di Computational REST.

4.3.1 I principi CREST

Vale la pena, prima di presentare la struttura del framework CREST-Erlang ed illustrarne l'implementazione (capitolo 5), riprendere i principi dell'architettura CREST, così come sono stati descritti in precedenza, e verificare se e come questi vengono ripresi nel nuovo framework, ed eventualmente quali compromessi si sono trovati nell'implementarli:

1. L'astrazione chiave dell'informazione è la *computazione*, specificata univocamente tramite un URL

CREST definisce due operazioni basilari: *remote* e *spawn*, come abbiamo visto; nel secondo caso, la computazione viene installata e deve rispondere ad un URL, cui deve essere associata una mailbox che ne gestisca le relative richieste. Come illustrato in precedenza, processi e mailbox sono gestiti nativamente da Erlang, e l'URL viene costruito associando a ciascuna computazione l'indirizzo del peer ed un codice univoco (UUID) generato nell'istante in cui si installa la computazione stessa.

Erlang consente l'invio di chiusure, tuttavia richiede che con la chiusura stessa venga inviato anche il codice intermedio (bytecode) in cui tale chiusura

è definita; il core del linguaggio offre funzionalità di caricamento e ricezione di tale bytecode dinamicamente durante la fase di runtime, perciò anche questa caratteristica è realizzabile.

Erlang non supporta nativamente le continuazioni.

2. Ciascuna risorsa è un programma, una chiusura, una continuazione o un ambiente di binding, corredato da metadati che descrivano ciascuna di queste entità

L'invio di parametri con cui invocare una determinata computazione non crea alcun tipo di problema: il protocollo HTTP prevede la presenza di tali parametri, e questi vengono tradotti in tuple e liste nel momento in cui il server Web riceve la richiesta e la invia al processo che deve gestirla.

La composizione di computazioni richiede che ciascuna di queste conosca l'indirizzo in cui trovare le parti che la compongono, anche se queste dovessero risiedere all'interno del medesimo peer.

3. Tutte le interazioni sono libere dal contesto

Le informazioni di stato possono naturalmente essere incluse nella chiusura o parametrizzate ed inviate assieme alla chiusura stessa (o all'atto di invocazione della stessa).

4. Solo poche operazioni basilari sono disponibili, ma è incoraggiato lo sviluppo di nuove operazioni legate alle singole risorse

Come prima, nuove operazioni possono essere installate su un server tramite *spawn*, tuttavia chiunque voglia utilizzare tali operazioni deve conoscerne l'indirizzo cui rispondono.

Si può pensare ad un'evoluzione del framework, in cui vengano introdotti meccanismi di discovery di servizi e delle loro capabilities.

5. E' promossa la presenza di intermediari

Gli intermediari sono peer che si frappongono fra quelli già esistenti; Erlang non ha alcun problema in questo, poiché la comunicazione avviene tramite protocollo HTTP e, di conseguenza, la trasmissione può essere soggetta a meccanismi di caching piuttosto che di ispezione.

4.3.2 I vincoli architetturali CREST

Come per i principi, anche i vincoli architetturali [16], presentati in precedenza come conseguenze per l'architettura dei peer (pagina 26), possono essere analizzati e verificate le caratteristiche che assumono nella nuova implementazione:

1. *Nomi*: l'unicità degli URL è garantita dalla presenza dell'indirizzo e dell'ID univoco per ciascuna operazione di *spawn* o *remote*; le caratteristiche dell'URL, tuttavia, cambiano in parte rispetto a quanto dichiarato dall'implementazione di riferimento, e la prossima sezione descrive come funzionerà la nuova struttura;

2. *servizi*: la parametrizzazione di un servizio avviene tramite parametri inviati durante l'invocazione di quel servizio o tramite la manipolazione delle variabili della chiusura;

3. *tempo*: anche in Erlang è possibile avere chiusure che mantengono uno stato, così come servizi i cui risultati cambiano nel tempo, a seconda dei parametri ricevuti;

4. *stato*: la parallelizzazione avviene automaticamente: due processi non possono avere memoria condivisa, quindi non c'è necessità di sincronizzazione e tutti i problemi che questa può portare;

5. *computazione*: una computazione definita tramite una chiusura può essere invocata più volte; Erlang non ha un meccanismo di definizione di una continuazione, quindi questa modalità di trasmissione del codice non è supportata;

6. *trasparenza*: il framework CREST implementato in Erlang utilizza HTTP, di conseguenza gli intermediari (ad esempio i meccanismi di caching) possono essere ancora utilizzati; eventuali meccanismi di inspection possono effettuare operazioni di reverse del bytecode trasmesso nelle operazioni di *spawn* e *remote*;

7. *migrazione e latenza*: lo spostamento di chiusure verso peer con maggiori capacità di computazione può avvenire senza problemi, anche se è necessario stabilire come un peer possa essere a conoscenza di queste informazioni; l'aggiunta di timestamp o firme crittografiche può essere fatta, al momento il framework non supporta queste caratteristiche.

4.3.3 Struttura degli URL

La struttura degli URL varia in parte nell'implementazione Erlang rispetto a quanto dichiarato nei documenti di presentazione di CREST: se infatti l'invocazione di un servizio *spawn* rimane la medesima, in particolare l'esempio

```
http://www.example.com/mailbox/
60d19902-aac4-4840-aea2-d65ca975e564
```

nel progetto corrente diviene

```
http://www.example.com/crest/
60d19902-aac4-4840-aea2-d65ca975e564
```

la richiesta di una delle due operazioni, che nell'originale ad esempio avviene come

```
http://server.example.com/(if(defined?'word-count)(word-count
(GET"http://www.yahoo.com/"))
```

ora avviene tramite una chiamata HTTP POST al peer ricevente, con la chiusura, il bytecode del modulo e gli eventuali parametri codificati nel corpo della chiamata, come mostrato in figura 4.2 e nell'appendice C (pagina 75), dove è possibile vedere la chiamata HTTP POST di un'invocazione del metodo *spawn* con una chiusura di test.

POST request to /crest/spawn

type	name	value
body	module	spawn_test_1
body	binary	FOR1%00%00%0BXBEAMAtom%00%00%00%DC%00%00%0...
body	filename	%2Fhome%2Falex%2FDocumenti%2FPolitecnico%2Fworksp...
body	code	%83p%00%00%00%9B%00%14%D9F%8AM%12%C1G%97%...

Figura 4.2: Parametri dell'invocazione di spawn

Si è deciso, pertanto, che le due operazioni di *remote* e di *spawn* in fase di installazione di un servizio siano invocate *solamente* tramite POST, mentre l'invocazione di un servizio precedentemente installato avvenga tramite GET o POST, in base alla decisione dello sviluppatore di quel servizio.

Si è ritenuto opportuno utilizzare tale modalità HTTP anche se è un'operazione dichiaratamente non idempotente, quindi in parte contraria alla filosofia di utilizzo di Computational REST, poiché maggiormente adatta alla trasmissione di contenuti che modificano lo stato del server (un nuovo servizio viene installato) e che richiedono di trasmettere un corpo del messaggio ingombrante (a causa della presenza non solo della chiusura ma anche del bytecode).

Il prossimo capitolo descriverà nel dettaglio l'implementazione e l'applicazione demo utilizzata per testare il nuovo framework.

Capitolo 5

Realizzazione del caso di studio

5.1 Architettura generale del framework

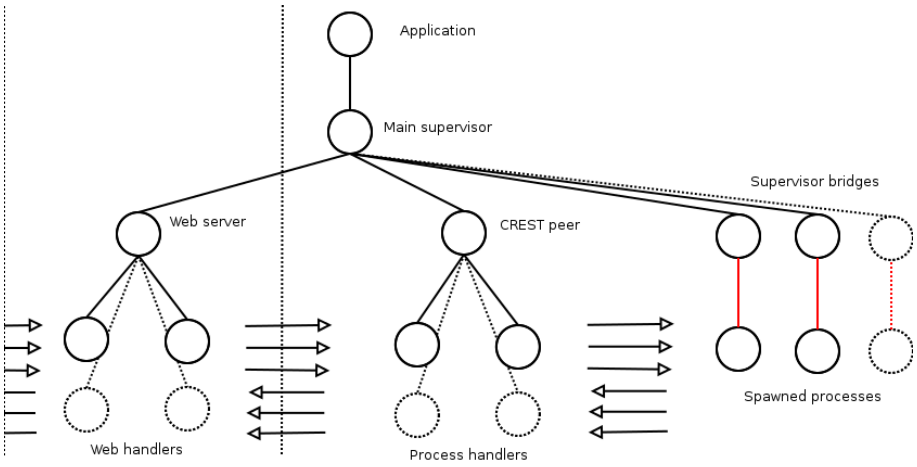


Figura 5.1: Architettura del framework Computational REST - Erlang

La figura 5.1 mostra una parte dell'architettura del framework CREST - Erlang[1]: nella parte sinistra dell'immagine le frecce indicano la trasmissione con i client, i quali si interfacciano con i processi dedicati alle singole richieste e lanciati dal processo server principale (grazie al *behaviour* definito da MochiWeb); questi processi dedicati gestiscono tutte le comunicazioni del server, ed utilizzano un modulo di routing per dirottare le chiamate CREST verso il peer server, il quale a sua volta le gestisce internamente se sono del tipo

```
https://somehost.somedomain/crest/spawn
```

[1] A pagina 61 è possibile vedere la panoramica completa.

```
https://somehost.somedomain/crest/remote
```

oppure le dirotta verso i processi che gestiscono le computazioni, se gli URL sono del tipo

```
http://somehost.somedomain/crest/
886f8a80-b1e0-47c3-9390-8cadc35f5701
```

Il processo peer, per effettuare queste operazioni, utilizza alcuni processi figli, in particolare per eseguire le chiamate che richiedono una risposta da parte delle computazioni, risposta che naturalmente può impiegare del tempo ad essere elaborata; in questo modo, invocazioni multiple che non cambiano lo stato interno del peer possono essere gestite contemporaneamente, esattamente come nel caso del server Web.

I processi riceventi eseguono le proprie computazioni e restituiscono un risultato lungo il percorso inverso. Questi processi sono monitorati da supervisori singoli, i quali a loro volta rispondono al supervisore generale, di fatto generando una gerarchia in grado di reagire ad eventuali problemi ed a registrare tali errori nei log del framework.

Lo scheletro dell'applicazione è stato generato utilizzando *MochiWeb*, ed ai moduli già presenti in grado di gestire la trasmissione di pagine Web statiche si sono aggiunti i moduli di gestione del protocollo Computational REST; il framework perciò consta dei seguenti blocchi:

- *crest_app*, il modulo principale di gestione delle dipendenze e di avvio del programma; è codice autogenerato.

- *crest_sup*, il supervisore radice dell'albero dei processi: qualunque altro sottoprocesso viene agganciato direttamente a lui o ai suoi figli, ed in presenza di errori è l'ultimo gestore delle eccezioni. Avvia i sottomoduli principali.

- *crest_web*, implementa il *behaviour gen_server* e riceve le richieste HTTP che arrivano all'applicazione; trasmette direttamente le eventuali pagine statiche e intercetta le comunicazioni CREST, che vengono girate al router.

  Questo modulo ha un suo gemello, *crest_web_ssl*, che supporta la gestione delle chiamate HTTPS; il funzionamento è sostanzialmente analogo a questo, tuttavia esso serve solamente richieste di *spawn* e *remote*, come verrà descritto più avanti.

- *crest_router*, smista le chiamate CREST verso il gestore del peer, invocandone le funzionalità di installazione o di esecuzione di una richiesta.

- *crest_peer* (pagina 65), implementa anch'esso il *behaviour gen_server*, mantenendo nello stato l'elenco dei processi installati; si occupa di eseguire le due operazioni di *spawn* e *remote*, di cancellare i processi non più necessari e di elencare i processi attivi.

- *crest_spawn* (pagina 70), implementa il *behaviour supervisor_bridge*, e si occupa di eseguire l'operazione di creazione di un nuovo processo.

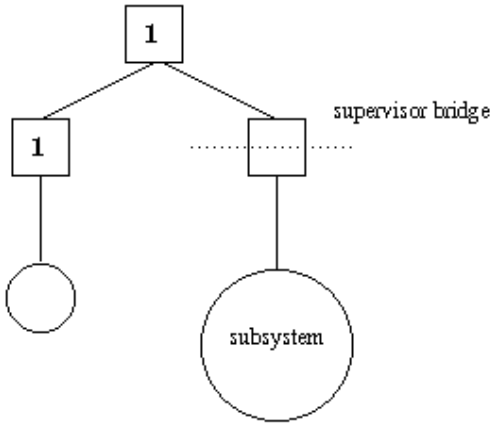


Figura 5.2: Behaviour supervisor_bridge

  Un supervisor_bridge, mostrato in figura 5.2[2], funge da ponte tra un sistema già sotto supervisione ed un sottosistema esterno, che non implementa alcun comportamento predefinito di Erlang; questo è il caso ad esempio dei processi che eseguono funzionalità installate tramite le operazioni di CREST: in questo caso, il bridge lancia il processo supervisore, agganciato all'albero di processi, lancia il processo da gestire e registra le attività di quest'ultimo, in particolare l'eventuale crash o lo spegnimento controllato. Non è attualmente in grado di riavviare un sottoprocesso che abbia subìto degli errori.

- moduli minori che implementano funzioni di utilità generale.

Come detto, le chiamate di tipo Computational REST sono indirizzate tramite il router verso le funzionalità corrette del peer; le chiamate di

[2]Tratta dalla documentazione di Erlang.

tipo *remote* sono di fatto una contrazione di una chiamata *spawn*, seguita immediatamente da un'invocazione del servizio e dalla sua cancellazione. [3]

L'installazione di un servizio comporta la creazione di due processi: quello che effettivamente esegue la chiusura ed il suo supervisore ponte, che a sua volta si interfaccia al supervisore radice; al chiamante viene restituita la chiave univoca cui risponde il nuovo servizio. Una chiamata successiva a quel servizio, in GET o in POST a seconda di quanto deciso dallo sviluppatore, produrrà l'invocazione di quella chiusura e la restituzione del valore al client.

Sta alla chiusura stessa gestire eventuali errori di esecuzione e di attendere nuovi messaggi al termine della singola esecuzione, di fatto il rilanciare la funzione stessa.

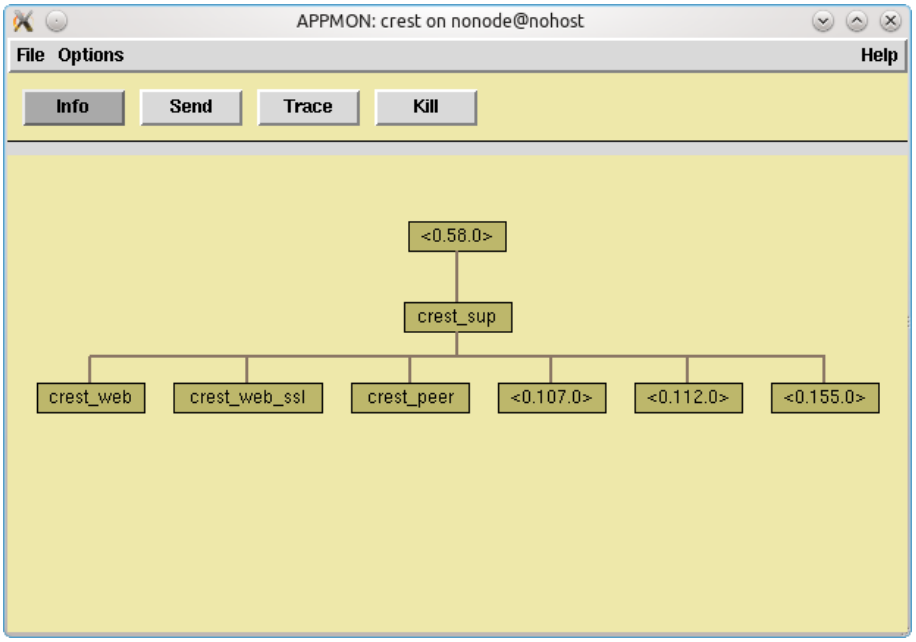


Figura 5.3: Processi in esecuzione in un peer

La figura 5.3 mostra uno snapshot dell'esecuzione di un peer, con i processi attivi: nella parte alta sono presenti l'applicazione ed il supervisore radice, il quale controlla il server peer ed i due server Web (in chiaro e criptato).

Sono presenti inoltre tre supervisori ponte, i quali a loro volta controllano ciascuno un processo che esegue una chiusura (i processi semplici non sono visualizzati in questa figura, quindi non è visibile neppure il pool di processi in attesa di chiamate per ciascuno dei due server Web); in questo momento, perciò, il peer CREST ha ricevuto tre richieste di *spawn*.

[3] Le figure A.2 e A.3 (pagina 62) mostrano i diagrammi di sequenza delle chiamate tra i moduli presentati per entrambe le richieste di operazioni, e sottolineano tale differenza.

La figura 5.4 mostra invece la parte di gestione delle chiamate dei moduli *_web*: ciascuno ha attivo un pool di sottoprocessi in attesa di richieste da parte dei client, e non appena uno di questi inizia a gestire una richiesta, un altro viene lanciato al suo posto, così da avere sempre un pool pronto ad accettare connessioni.

Sono stati fatti alcuni test di invocazioni parallele e contemporanee al server, da alcune centinaia a migliaia, ed il server stesso è sempre stato in grado di gestire tutte le chiamate senza alcun timeout, seppur su una macchina non provvista di hardware server.

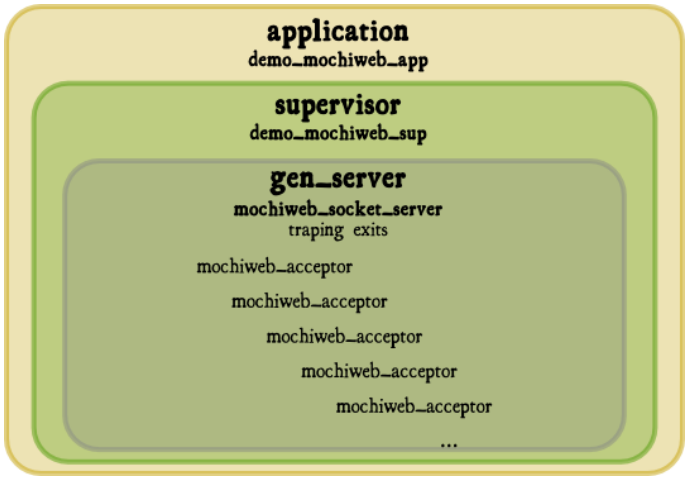


Figura 5.4: Struttura dei processi MochiWeb

Il listato a pagina 72 mostra un esempio di modulo che definisce una chiusura in grado di ricevere parametri ed eseguire funzionalità, gestendo eventuali errori interni e comunicandoli al chiamante. Il processo di gestione delle chiusure si aspetta come risultato una tupla che specifichi il tipo di contenuto (parametro *Content-Type* di HTTP) ed il contenuto stesso da allegare come corpo del messaggio di risposta.

La versione attuale del framework supporta sia HTTP che HTTPS, grazie alle funzionalità introdotte dalle ultime versioni di MochiWeb, a loro volta basate sull'introduzione di un supporto SSL integrato ed affidabile in Erlang (a partire sostanzialmente dalla release *R14A*).

L'applicativo utilizza due istanze separate di server Web, quindi due distinti processi di gestione delle richieste provenienti dai client: la prima risponde sulla porta 80 e la seconda sulla 443, utilizzando certificati e chia-

vi create ad-hoc per il peer in questione; conseguentemente, si è deciso di trasferire su SSL le chiamate *spawn* e *remote*.

Ciascuna operazione di trasferimento codice su SSL richiede, inoltre, una mutua autenticazione tra il peer che assume il ruolo di client e quello che assume il ruolo di server: questo permette non solo al chiamante di verificare l'autenticità del server, ma a quest'ultimo di verificare se il client ha un certificato correttamente firmato dalla CA corrente, aspetto piuttosto importante dato che sarà proprio il server ad eseguire il codice che gli è stato inviato, ed uno dei problemi tipici di questi sistemi, come peraltro già sottolineato in precedenza, è cercare di avere qualche tipo di garanzia su tale codice.

Al momento le uniche operazioni disponibili con SSL sono quelle CREST, se tuttavia si vuole rendere disponibile anche alle singole computazioni tale layer di sicurezza, magari con differenti certificati, è necessario frapporre un modulo tra i client ed il peer CREST, utilizzando il progetto *Nginx* [5], il quale offre una configurazione puntuale dei singoli URL o famiglie di URL con i dettagli di sicurezza, i certificati ed i dettagli sulla necessità o meno della mutua autenticazione, in maniera più granulare di quanto non facciano i meccanismi standard di Erlang.

5.1.1 Il manager

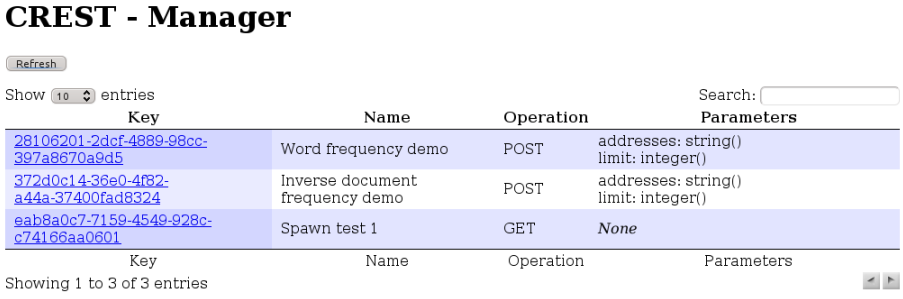


Figura 5.5: Il manager di processi

La versione attuale del framework contiene anche un manager dei subpeer, come mostrato in figura 5.5: al momento, esso è utile per verificare il numero di processi in esecuzione ed alcune informazioni che li riguardano; si può prevedere tuttavia di aumentarne le funzionalità, ad esempio dando la possibilità di eliminare chiusure o di lanciarne direttamente da qui, in modo simile al manager di Apache Tomcat, o di richiedere ai moduli installati di essere in grado di fornire specifiche della funzione che implementano.

Allo stato attuale, alle chiusure da installare si richiede di saper gestire richieste aventi come parametro le coppie *param=name*, *param=operation* e *param=parameters*, le quali restituiscono il nome del servizio, la modalità di accesso (GET/POST) e quali attributi e di che tipo devono essere inviati con la richiesta (il tipo è specificato secondo la sintassi definita per la documentazione di tipi in Erlang, ma è naturalmente possibile utilizzare qualunque altra convenzione, dato che si tratta di testo semplice); lo stesso esempio B.3 mostra come gestire queste richieste.

5.2 La demo

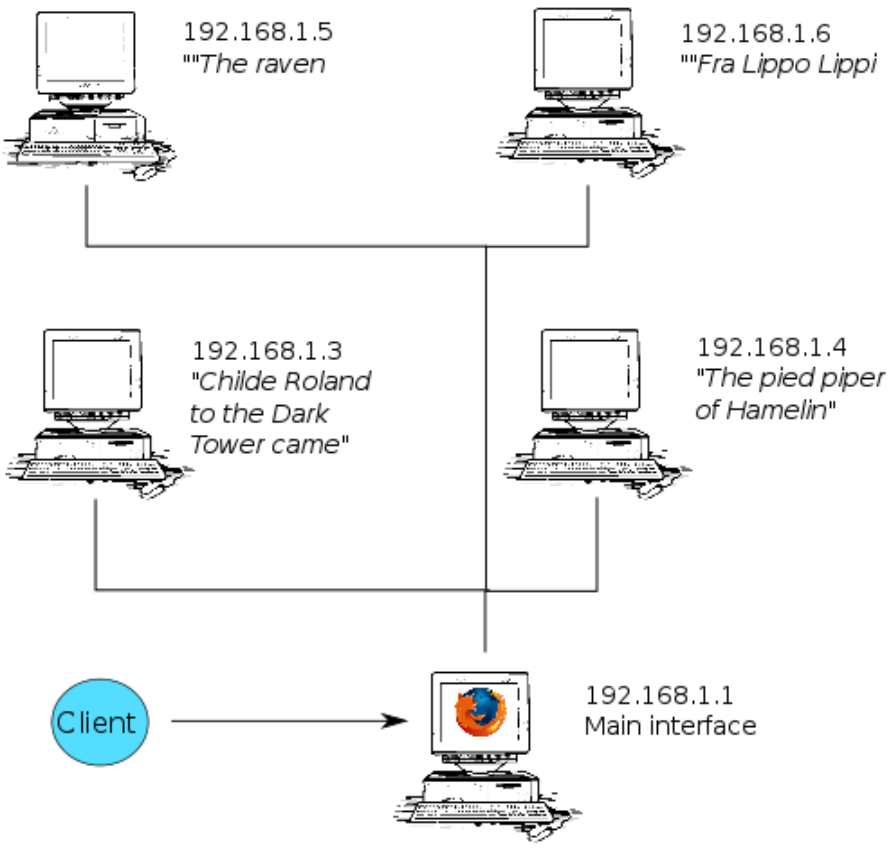


Figura 5.6: Schema della rete locale simulata dalla demo

L'applicazione demo, costruita per testare il funzionamento e le potenzialità del framework progettato, opera una sorta di "mashup" strutturato nel seguente scenario: si ipotizza di essere posizionati in una rete di compu-

ter, ciascuno avente installato il framework e di conseguenza ciascuno agente come peer CREST; uno dei computer della rete funge da interfaccia verso l'esterno, mentre gli altri componenti contengono degli archivi documentali di qualche tipo.

Nel momento in cui si vuole offrire un nuovo tipo di servizio verso i clienti che utilizzano il terminale di accesso, ad esempio un nuovo tipo di analisi da svolgere sui documenti presenti nella rete, anziché dover aggiornare tutte le singole installazioni si utilizzano le capacità di Computational REST, caricando le nuove funzionalità solamente sul computer-interfaccia.

Tale computer offre agli utenti un sito Web che permette di selezionare quale servizio sfruttare, ed in base a questa scelta viene quindi installata nel peer locale la chiusura in grado di elaborare il risultato, e questa a sua volta invia dinamicamente agli altri componenti della rete una computazione che compia l'analisi vera e propria sui documenti. Quando ciascun componente ottiene un risultato, lo invia al computer di partenza, il quale ha come scopo assemblare i singoli risultati e mostrare all'utente finale, tramite il sito Web, il risultato complessivo.

La figura 5.6 mostra un esempio di rete: i PC da 1.2 a 1.5 contengono ciascuno il testo di un poema in lingua inglese, mentre il PC 1.1 è quello che trasmette la chiusura agli altri, assembla i risultati e li visualizza sul sito Web.

La demo offre la possibilità di svolgere alcune analisi su questi testi:

- frequenza dei termini, mostrata singolarmente per ciascun documento, è un semplice conteggio della frequenza delle parole[4];
- frequenza dei termini e frequenza inversa dei documenti [33], un'altra funzionalità separata per documento, calcola un indice logaritmico a partire dalla frequenza dei termini nel singolo documento rispetto ai documenti in cui essa compare, operando quindi un'elaborazione successiva a partire dai singoli risultati;
- coseno di similitudine [36], a partire anch'esso dalla frequenza dei termini confronta ciascuna coppia di documenti e stabilisce quanto i componenti della coppia siano simili tra loro, combinando quindi anche in questo caso i risultati parziali; un esempio di esecuzione è mostrato in figura 5.7;
- frequenza dei termini con stato: la computazione è analoga a quella precedentemente descritta, tuttavia questa chiusura mantiene uno

[4]Un esempio di esecuzione è mostrato in figura A.4 a pagina 63.

stato interno, ovvero la raccolta delle parole più frequenti raccolte dagli indirizzi passati fino a quel momento, ed è possibile aggiungerne dinamicamente di nuovi ed aggiornare tale stato.

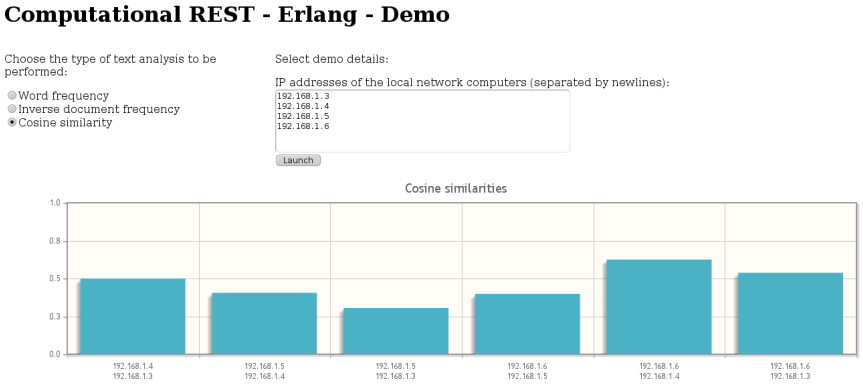


Figura 5.7: Cosine similarity

La composizione di servizi è uno dei punti positivi riscontrati dagli autori di CREST verso il Web odierno, di conseguenza impiegare il framework in un'applicazione che esegua anch'essa una composizione di servizi mi è sembrato un percorso piuttosto naturale; al contempo, la demo dimostra il vantaggio di eseguire codice mobile: le funzionalità vengono inviate direttamente ai device coinvolti, ed eventuali cambiamenti possono essere propagati nel momento in cui i servizi sono richiesti, senza dover impiegare strategie di aggiornamento su vasta scala.

Non solo: la seconda, la terza e la quarta funzionalità sono costruite a partire dai risultati della prima funzionalità, perciò anziché replicare il codice, queste ultime si appoggiano direttamente al primo servizio, installandolo ed invocandolo, ed eseguendo le loro computazioni ulteriori direttamente sui risultati che la prima demo normalmente restituisce al client; questo dimostra come lo sfruttamento di codice esistente sia possibile, pur dovendo lo sviluppatore conoscere quali servizi sono già disponibili all'interno dei peer CREST.

La quarta funzionalità, infine, è stata creata per mostrare come sia possibile avere chiusure con uno stato interno, come sia possibile aggiornare tale stato e, a scelta dello sviluppatore, se sia possibile accedere a tale stato: nella demo, ad esempio, un'invocazione del servizio senza alcun indirizzo porta alla visualizzazione dello stato corrente, quindi delle frequenze di parole raccolte fino a quel momento.

5.3 Confronto tra framework

Dopo aver presentato le funzionalità del framework Erlang e come esse siano state sfruttate nella demo, è ora possibile effettuare un confronto su alcuni aspetti delle due implementazioni, elencando pregi e difetti di quella qui presentata nei confronti di quella originale.

I due progetti

Il framework originario, per riprendere alcuni concetti visti in precedenza, è stato implementato in Scheme; le sue potenzialità sono state presentate attraverso un'applicazione demo, in particolare si tratta di un feed reader condiviso: il peer CREST mantiene alcune computazioni con stato che gestiscono un lettore di feed scaricati da Internet, permettendo a diversi client Web standard (comuni browser) di accedere a tali feed ed averne la medesima visualizzazione, con la possibilità di selezionarne di nuovi ed automaticamente avere tutti i client sincronizzati con le nuove viste.

Tale demo permette inoltre di duplicare la panoramica corrente, con la conseguente duplicazione delle computazioni sottostanti e la generazione di diverse istanze di ciascuna di esse; ognuna è ottenuta tramite *spawn*, quindi è di fatto un *subpeer*, secondo il lingaggio definito dagli autori.

Il framework qui presentato è stato implementato in Erlang; l'applicazione demo non è analoga a quella originaria, in questo caso si tratta infatti di un sistema per effettuare operazioni di text mining (prettamente sintattico) su testi distribuiti su diversi dispositivi; anche in questo caso alcune computazioni sono in grado di mantenere uno stato interno, anche se non tutte, a differenza della demo originaria.

Funzionalità del framework

Il framework Erlang permette di effettuare operazioni analoghe a quelle originarie: per quanto sia possibile desumere dalla demo visionata del progetto Scheme, di cui non esiste una descrizione tecnica puntuale, la possibilità di lanciare computazioni tramite le due operazioni di Computational REST e la possibilità di avere uno stato sono le caratteristiche principali e sono state mantenute.

La mancanza principale di Erlang nei confronti di Scheme è il supporto per le continuazioni, come descritto in precedenza, di conseguenza non esistono facilitazioni immediate per l'interruzione di una computazione o per la sua duplicazione, tuttavia quest'ultima caratteristica può senz'altro essere

implementata anche con Erlang, trattandosi in questo caso di trasferire lo stato corrente delle chiusure; lo sviluppatore delle singole funzionalità delle chiusure deve tuttavia poter prevedere questa funzionalità.

Per il resto, le chiusure permettono comunque l'invio di una computazione ed il mantenimento di dati tra un'invocazione e l'altra; Erlang necessita inoltre di inviare il bytecode del modulo di definizione della chiusura, tuttavia il framework Scheme non supporta la trasmissione di codice tra diversi peer, di conseguenza non è possibile stabilire se sia necessario o meno anche per le continuazioni l'invio del codice sorgente in aggiunta alla continuazione stessa (in generale, dipende dall'implementazione dell'interprete in uso).

I concetti di *subpeer* e di *fractalware*, introdotti in un secondo momento dagli autori di CREST, non sono stati qui replicati: il peer è unico, infatti, e le singole computazioni non gestiscono a loro volta le due operazioni di *spawn* e *remote*; tuttavia, l'esistenza di un albero di supervisione e di un modulo dedicato al routing delle chiamate, può permettere di inserire ciascuna computazione in un contesto che sia in grado a sua volta di lanciare nuovi processi, ricordando comunque che detti meccanismi di supervisione sono pensati per gestire correttamente sistemi che rispettino i *behaviour* di OTP, non processi generici, quindi tale albero di supervisione potrebbe risultare particolarmente appiattito su meno livelli rispetto alla teoria.

Implementazione del framework

Il framework Erlang implementa correttamente i principi ed i design architetturali richiesti da Computational REST, come descritto nel capitolo 4; non esistendo un riferimento puntuale per alcuni aspetti, alcune decisioni sono state prese in modo autonomo, ad esempio la scelta di trasmettere una chiusura tramite l'operazione *HTTP POST*, tuttavia in queste decisioni si è cercato di restare coerenti con gli obiettivi e le funzionalità che CREST vuole offrire.

Il codice prodotto sfrutta gran parte delle facilitazioni riguardo la comunicazione tra processi che il linguaggio offre, evitando perciò la necessità di reimplementare alcune delle funzioni base, come ad esempio la nozione di processo o di mailbox, che invece l'implementazione Scheme di riferimento ha avuto necessità di ricreare o di utilizzare interfacciandosi con librerie Java, operazione possibile grazie all'impiego di un interprete compatibile con la Virtual Machine di questo linguaggio.

In Erlang invece, come già spiegato precedentemente, si è trattato di mantenere la struttura di comunicazione del linguaggio ed incapsulare le primitive di invio e ricezione dei messaggi su HTTP(S) anziché sul protocollo standard TCP utilizzato dall'interprete, e di riscrivere la normale operazione di invio di Erlang come *spawn* (essendo *remote* di fatto un caso particolare della precedente).

Ove necessario, OTP offre inoltre una libreria Java per la creazione di nodi compatibili con Erlang, così da integrare eventuali librerie esterne, oltre alla possibilità di scrivere wrapper in C/C++.

Da aggiungere, infine, che Erlang supporta il mantenimento di al più due versioni in memoria del medesimo modulo: questo comporta il fatto che se richieste successive di *spawn* inviano più volte il medesimo bytecode, e questo viene ricaricato in memoria ad ogni passaggio, tutti i processi che sfruttano le chiamate alla penultima versione più vecchia verranno interrotti senza possibilità di recupero.

Performance

Da un punto di vista di prestazioni del framework, non esiste un riferimento per l'implementazione in Scheme che sia in grado di esprimere ad esempio il carico che il peer sia in grado di servire; Erlang, invece, è noto per le elevate prestazioni soprattutto su sistemi distribuiti o multicore, se le primitive di creazione e gestione dei processi sono adeguatamente sfruttate.

L'implementazione qui presentata cerca, perciò, di seguire tali specifiche nei moduli creati, suddividendo il carico tra processi che eseguano le computazioni più complesse o potenzialmente bloccanti per il singolo peer; al contempo, la libreria Web in uso, Mochiweb, è implementata per gestire anche un carico notevole ed alcuni test sono stati resi pubblici [1], mostrando come con poche modifiche si possano raggiungere prestazioni anche molto elevate.

Sicurezza

I lavori di presentazione di CREST [17], come descritto nei capitoli precedenti, illustrano alcune tecniche utilizzabili per diminuire i rischi di sicurezza legati al codice mobile, senza tuttavia fornire alcun dettaglio implementativo al riguardo, salvo la citazione della sandbox fornita dalla VM di Java.

Erlang in questo aspetto pecca sicuramente in maniera decisa, dato che numerosi meccanismi di sicurezza mancano completamente o hanno imple-

mentazioni molto giovani, e segno di questo è stato l'introduzione di un layer SSL funzionante solamente pochi mesi fa; il framework qui presentato sfrutta di fatto il solo SSL come meccanismo di sicurezza, in particolare per la mutua autenticazione tra i peer: la trasmissione di codice richiede infatti sia al peer mittente che a quello ricevente di avere un certificato valido e firmato da una CA fidata (nella demo, la CA è stata creata ad hoc per motivi di test), questo affinché ciascuno dei due possa avere una minima traccia di affidabilità della controparte.

Alcune esistenti tecniche di ispezione del codice sono state presentate nel capitolo 2, sono replicabili e migliorabili in eventuali future evoluzioni del framework stesso.

Capitolo 6

Conclusioni

Questa tesi ha descritto l'implementazione di un framework in grado di supportare lo stile architetturale Computational REST, il quale riprende alcuni dei concetti di "codice mobile" e li trasferisce sul Web; secondo gli autori originari questi concetti sono maggiormente adatti al mondo di Internet che esiste oggi, e sicuramente diverse applicazioni del cosiddetto "Web 2.0" sfruttano già meccanismi simili.

Ciò che differenzia questo lavoro rispetto a quello originario presentato in [17] è il linguaggio scelto per la realizzazione: si è infatti optato per Erlang, un linguaggio funzionale orientato alla comunicazione, il quale offre diverse facilitazioni per la creazione di sistemi distribuiti e comunicanti tra loro. Diverse di queste facilitazioni vengono introdotte anche in CREST, e si è pensato di sfruttarle per creare un framework più semplice come struttura e maggiormente performante, e naturalmente in grado di offrire le medesime caratteristiche.

La creazione di questo framework ha avuto esito positivo: i capitoli precedenti hanno mostrato come le richieste siano state mantenute ed i principi architetturali rispettati; le eventuali scelte autonome rispettano i requisiti, ed alcuni aspetti sono stati approfonditi e descritti più nel dettaglio rispetto al lavoro originario, ad esempio l'implementazione di specifici meccanismi di sicurezza.

Le conclusioni

La prima conclusione che possiamo trarre riguarda il linguaggio di programmazione scelto: Erlang è estremamente interessante in quanto in grado di offrire facilitazioni per la gestione di processi, una compattezza del codice tipica dei linguaggi funzionali, la capacità di gestire internamente gli aspetti

non funzionali in particolare dei server generici, la possibilità di sostituire codice "a caldo" e di aggiornare un applicativo senza doverlo fermare.

Il punto debole è la sicurezza, ed il giudizio sarebbe stato sicuramente peggiore se il rilascio delle ultimissime versioni non avesse mostrato che tali problematiche, mai affrontate fino ad oggi, si stiano finalmente rivedendo e si sta cercando di porvi rimedio; questo perciò fa ben sperare sulle evoluzioni che ci aspettano nei prossimi mesi.

Inoltre, il fatto che il linguaggio stesso sia nato per gestire apparecchiature e device generalmente con scarse capacità computazionali potrebbe permettere l'installazione di questo framework anche in device diversi dall'hardware tipico dei server, permettendo ad esempio il movimento di codice e l'installazione dinamica di servizi in dispositivi mobili.

La seconda conclusione riguarda naturalmente il framework Computational REST: diverse affermazioni discusse nella tesi di CREST rispecchiano effettivamente il fatto che gli strumenti che operano oggi nel Web sono evoluzioni costruite su linguaggi e protocolli che in origine non erano stati pensati ad un uso così su larga scala: Internet si è evoluto in un modo non dico imprevedibile quanto piuttosto inaspettato, e di conseguenza un cambiamento nel paradigma che guida la creazione di applicazioni è sicuramente necessario.

Ciò detto, è allo stesso tempo chiaro come il paradigma di codice mobile necessiti di cambiamenti ancora più profondi; l'aspetto più particolare secondo me, ma forse al contempo a sua volta necessario per garantire la retrocompatibilità, è la trasmissione di bytecode utilizzando un protocollo come HTTP: se è vero da un lato che i principi REST sono interessanti e garantiscono la scalabilità delle applicazioni, al contempo HTTP è pensato per la trasmissione di contenuti e non di codice, non quantomeno con la frequenza che una Rete ampia che utilizzi CREST richiederebbe; l'utilizzo di un protocollo alternativo, eventualmente preso tra quelli già esistenti, potrebbe essere più adatto a questo scopo.

Io non sono in grado di dire oggi se Computational REST possa avere un futuro sicuro all'interno dello scenario del futuro Web, il cui discorso è sicuramente in parte più ampio e coinvolge concetti quali la semantica, tuttavia le potenzialità ci sono e la possibilità di avere framework in grado di supportare più linguaggi contemporaneamente (e non solo Erlang o Scheme) potrebbe essere un'ulteriore spinta per il suo sviluppo.

Sviluppi futuri

Per quanto riguarda, infine, i futuri sviluppi del framework qui presentato, essi coinvolgono l'utilizzo del Web server *Nginx* [5] come frontend a quello

in Erlang, così da permettere un'ulteriore spinta alle performance ed una maggiore configurabilità di SSL; altri aspetti riguardano la possibilità di aggiungere meccanismi di ispezione del codice stesso, l'inclusione del concetto di *subpeer*, così da implementare appieno le specifiche di CREST definite dagli autori originali.

Il manager delle computazioni può essere evoluto e permettere una effettiva gestione del peer e non un semplice monitoraggio del suo funzionamento; anche le richieste fatte alle singole computazioni possono non essere solamente limitate alla visualizzazione di informazioni generiche, quanto alla possibilità di modificare la computazione stessa in qualche modo.

Test di carico specifici per questo framework possono inoltre essere preparati, così da verificare effettivamente se l'implementazione riesce a scalare ed eventualmente intervenire per favorire questa capacità, che come mostrato in precedenza è sicuramente supportata dal linguaggio e dalle librerie in uso.

Bibliografia

[1] Erlang and mochiweb performances. `http://www.metabrew.com/article/a-million-user-comet-application-with-mochiweb-part-1`.

[2] log4erl. `http://code.google.com/p/log4erl/`.

[3] Mochi*. `http://www.mochibot.com/`, `http://en.mochimedia.com/`.

[4] Mochiweb. `http://github.com/mochi/mochiweb`.

[5] Nginx. `http://wiki.nginx.org/Main`.

[6] David Alan and David Alan Halls. Applying mobile code to distributed systems. Technical report, 1997.

[7] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, 2003.

[8] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.

[9] Lawrie Brown. Custom safety policies in safe erlang. In *ACISP '00: Proceedings of the 5th Australasian Conference on Information Security and Privacy*, pages 30–40, London, UK, 2000. Springer-Verlag.

[10] Dave Bryson and Steve Vinoski. Build your next web application with erlang. *IEEE Internet Computing*, 13:93–96, 2009.

[11] Francesco Cesarini and Simon Thompson. *Erlang Programming: A Concurrent Approach to Software Development*. O'Reilly Media, June 2009.

[12] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan

Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems: A research roadmap. pages 1–26, 2009.

[13] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna. Analyzing mobile code languages. In *MOS '96: Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet*, pages 93–110, London, UK, 1997. Springer-Verlag.

[14] Justin R. Erenkrantz. Architectural styles of extensible rest-based applications. In *Institute for Software Research, Report UCI-ISR-06-12*, 2006.

[15] Justin R. Erenkrantz, Michael Gorlick, Girish Suryanarayana, and Richard N. Taylor. From representations to computations: the evolution of web architectures. In *ESEC-FSE '07: Proceedings of the 6th joint meeting of the european software engineering conference and the 14th ACM SIGSOFT symposium on Foundations of software engineering*, pages 255–264, New York, NY, USA, 2007. ACM Press.

[16] Justin R. Erenkrantz, Michael M. Gorlick, and Richard N. Taylor. Crest: A new model for decentralized, internet-scale applications.

[17] Justin Ryan Erenkrantz. *Computational REST: a new model for decentralized, internet-scale applications.* PhD thesis, Long Beach, CA, USA, 2009. Adviser-Taylor, Richard N.

[18] Roy T. Fielding. *Architectural styles and the design of network-based software architectures.* PhD thesis, 2000.

[19] Matthew Daniel Fuchs. *Dreme: for life in the net.* PhD thesis, New York, NY, USA, 1995. Adviser-Perlin, Ken.

[20] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24:342–361, 1998.

[21] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering.* Prentice-Hall, Inc., 1991.

[22] Carlo Ghezzi and Giovanni Vigna. Mobile code paradigms and technologies: A case study. pages 39–49. Springer, 1997.

[23] M.M. Gorlick, J.R. Erenkrantz, and R.N. Taylor. The infrastructure of a computational web. Technical report, University of California, Irvine, May 2010.

[24] Rickard Green. Enhancing security in distributed erlang by integrating access control.

[25] Armstrong J. and Wiger U. Erlhive. `http://erlhive.sourceforge.net/`.

[26] Michael Kaminsky and Eric Banks. Sfs-http: Securing the web with self-certifying urls.

[27] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01*, pages 327–353.

[28] Alessandro Margara, Liliana Pasquale, and Alessandro Sivieri. Programming language support for dynamic software evolution. Articolo per il corso Argomenti avanzati di Ingegneria del Software, 2009.

[29] P. Oreizy and R. Taylor. On the role of software architectures in runtime system reconfiguration. *Configurable Distributed Systems, International Conference on*, 0:61, 1998.

[30] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 899–910, New York, NY, USA, 2008. ACM.

[31] K. Rikitake and K. Nakao. Application security of erlang concurrent system. In *Proceedings of IPSJ Computer Security Symposium 2008 (CSS2008)*, volume 2008, pages 253–258. IPSJ, 2008.

[32] K. Rikitake and K. Nakao. Ssh distribution transport on erlang concurrent system. In *Proceedings of IPSJ Computer Security Symposium 2009 (CSS2009)*, volume 2009, pages 117–122. IPSJ, 2009.

[33] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5):513 – 523, 1988.

[34] Dave Smith. Securing distributed erlang, 2008.

[35] Gerald Jay Sussman and Guy L. Steele. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11:405–439, 1998. 10.1023/A:1010035624696.

[36] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[37] Dimitris Vyzovitis. Mast: A dynamic language for programmable networks. Technical report, 2002.

[38] J. Zachary. Protecting mobile code in the world. *Internet Computing, IEEE*, 7(2):78 – 82, mar. 2003.

Appendice A

Figure aggiuntive

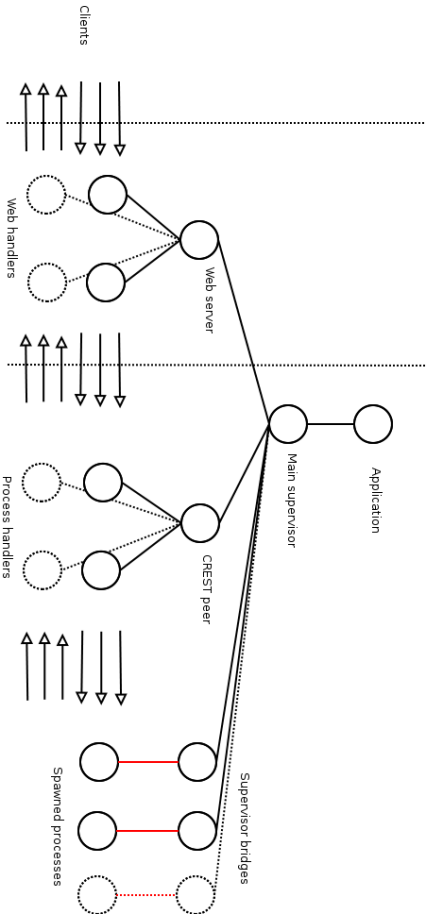


Figura A.1: Architettura del framework Computational REST - Erlang

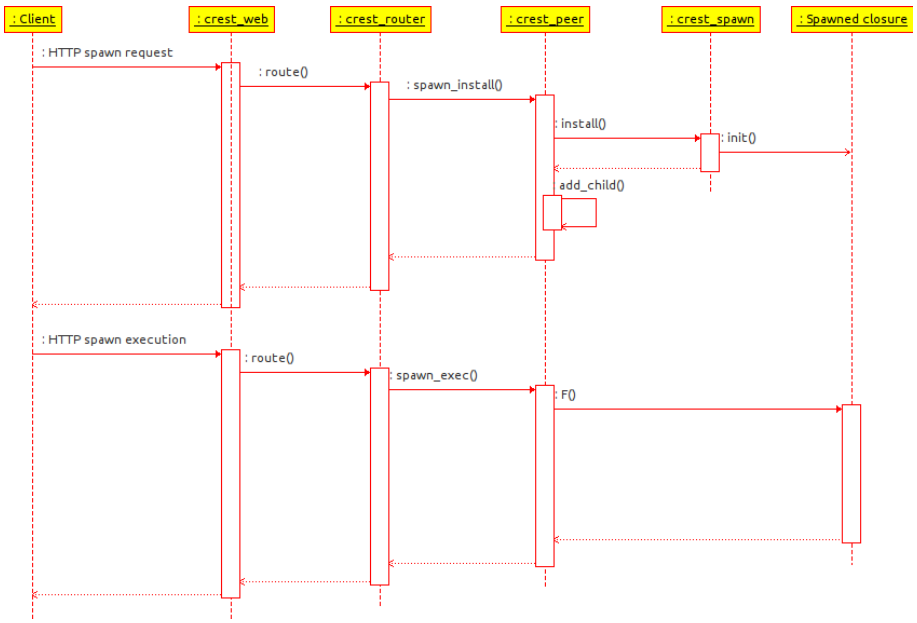


Figura A.2: Sequenza delle chiamate di tipo spawn

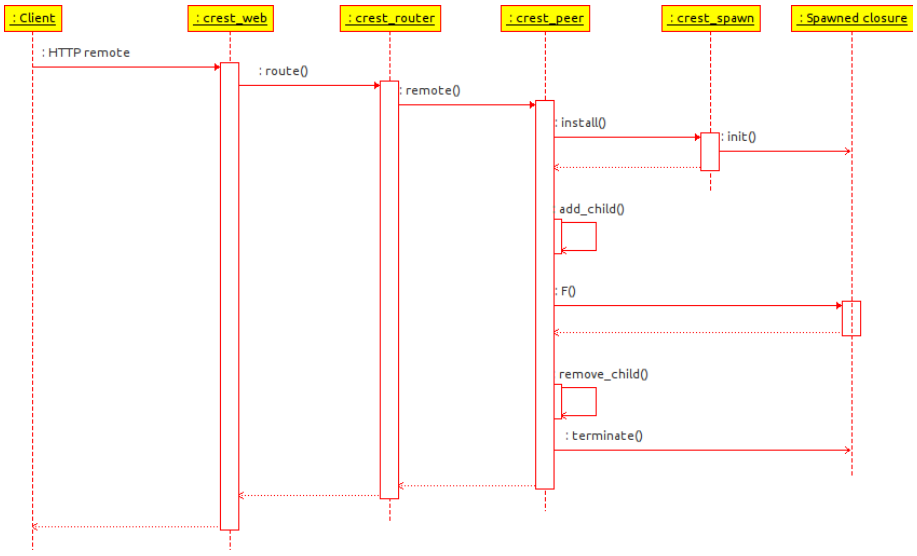


Figura A.3: Sequenza delle chiamate di tipo remote

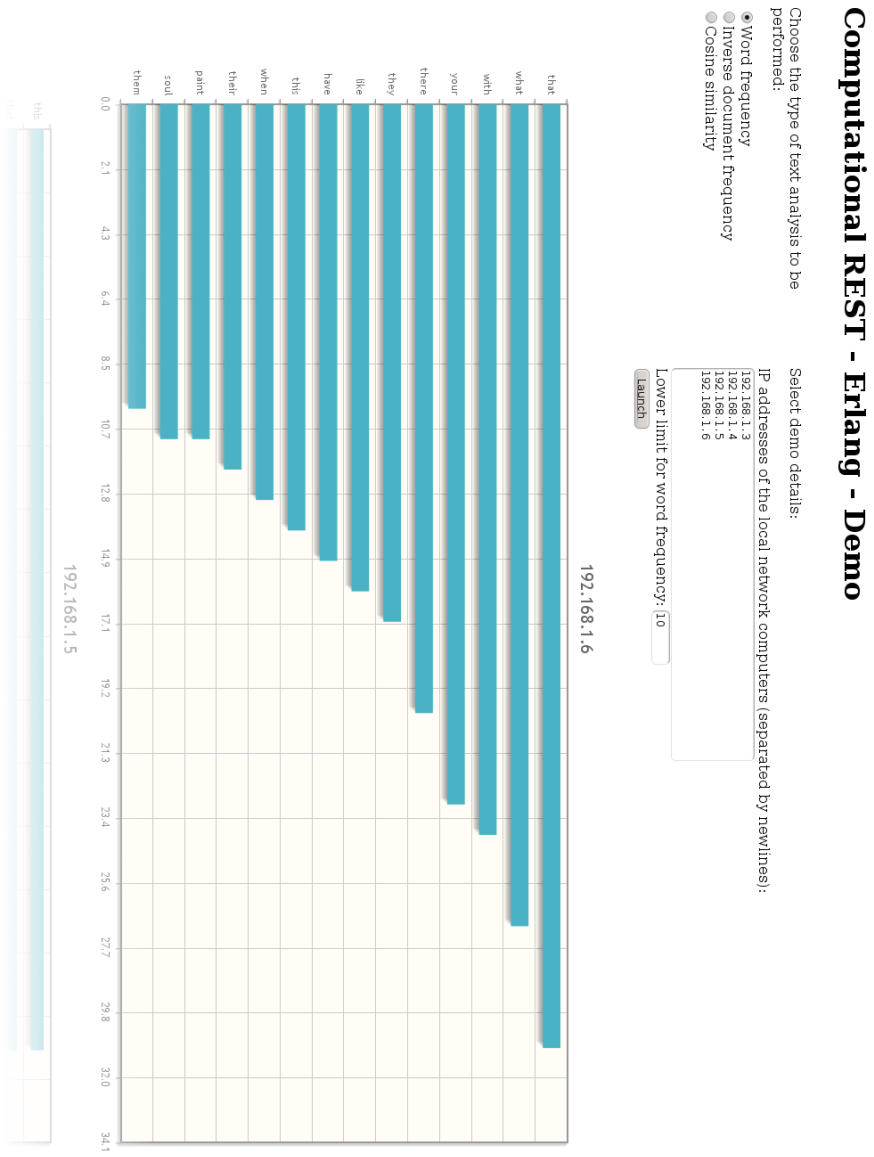


Figura A.4: Word frequency analysis (excerpt)

Appendice B

Codice del progetto (frammenti)

Listing B.1: CREST peer

```
%% @author Alessandro Sivieri <alessandro.sivieri@mail.
    polimi.it>
%% @doc Server module of a CREST peer.
%% It offers all the standard server start, stop and
    response
%% methods, plus the two specific CREST operations:
    spawn (split
%% into the installation and the execution parts) and
    remote.
%% All gen_server:call() operations are relegated to
    subprocesses,
%% so that the main server process is never blocked
    waiting for
%% an answer, and this allows this peer to call itself.
%% @copyright 2010 Alessandro Sivieri

-module(crest_peer).
-behaviour(gen_server).
-export([start/0, stop/0, spawn_install/1, remote/1,
    spawn_exec/2, add_child/2, remove_child/1, get_list
    /1]).
-export([init/1, handle_call/3, handle_cast/2,
    handle_info/2, code_change/3, terminate/2]).
```

```
%% External API

%% @doc Start this peer
%% @spec start() -> ok
start() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, []
        , []).

%% @doc Stop this peer
%% @spec stop() -> ok
stop() ->
    gen_server:call(?MODULE, stop).

%% @doc Install a new operation on this peer, using the
    given
%% parameters.
%% @spec spawn_install([{string(), any()}]) -> {string
    (), dictionary()}
spawn_install(Params) ->
    gen_server:call(?MODULE, {spawn, Params}).

%% @doc Execute an already installed operation,
    specified by the given
%% unique key.
%% @spec spawn_exec([Key], [{atom(), any()}]) -> {{ok,
    any()}, dictionary()} | {reply, {error}, dictionary
    ()}
spawn_exec([Key], Params) ->
    gen_server:call(?MODULE, {exec, Key, Params});
spawn_exec(Key, Params) ->
    gen_server:call(?MODULE, {exec, Key, Params}).

%% @doc Install and execute a new operation on this
    peer, and then delete it.
%% @spec remote([{string(), any()}]) -> {{ok, any()},
    dictionary()} | {{error}, dictionary()}
remote(Params) ->
    Key = gen_server:call(?MODULE, {spawn, lists:
        sublist(Params, 4)}),
```

```
    Answer = gen_server:call(?MODULE, {exec, Key, lists
        :sublist(Params, 5, length(Params))}),
    gen_server:cast(?MODULE, {delete, Key}),
    Answer.

%% @doc Add a new process to the internal server list,
    with the associated key.
%% @spec add_child(string(), pid()) -> dictionary()
add_child(Key, Pid) ->
    gen_server:cast(?MODULE, {add_child, Key, Pid}).

%% @doc Remove a child process from the internal server
    list, and terminate
%% it.
%% @spec remove_child(string()) -> dictionary()
remove_child(Key) ->
        gen_server:cast(?MODULE, {delete, Key}).

%% @doc Get a dictionary of responses from all childs,
    passing to all the given parameter;
%% the key is the child process UUID.
%% @spec get_list({string(), string()}) -> dictionary()
get_list(Param) ->
    gen_server:call(?MODULE, {list, Param}).

init(_Args) ->
    Spawned = dict:new(),
    {ok, Spawned}.

handle_call({spawn, Params}, From, Spawned) ->
        spawn(fun() -> handle_spawn(Params, From) end),
    {noreply, Spawned};
handle_call({exec, Key, Params}, From, Spawned) ->
        spawn(fun() -> handle_exec({Key, Params}, From,
            Spawned) end),
    {noreply, Spawned};
handle_call({list, Param}, From, Spawned) ->
        spawn(fun() -> handle_list(Param, From, Spawned
            ) end),
    {noreply, Spawned};
```

```
handle_call(_Request, _From, Spawned) ->
    {noreply, Spawned}.

handle_cast({add_child, Key, Pid}, Spawned) ->
    NewSpawned = dict:store(Key, Pid, Spawned),
    log4erl:info("Registered a new key ~p~n", [Key]),
    {noreply, NewSpawned};
handle_cast({delete, Key}, Spawned) ->
    case supervisor:terminate_child(crest_sup, Key) of
        ok ->
            supervisor:delete_child(crest_sup, Key),
            NewSpawned = dict:erase(Key, Spawned),
            log4erl:info("Deleted the key ~p~n", [Key])
                ,
            {noreply, NewSpawned};
        {error, not_found} ->
            {noreply, Spawned}
    end;
handle_cast(_Request, Spawned) ->
    {noreply, Spawned}.

handle_info(_Info, Spawned) ->
    {noreply, Spawned}.

code_change(_OldVsn, Spawned, _Extra) ->
    {ok, Spawned}.

terminate(_Reason, _Spawned) ->
    ok.

%% Internal API

handle_spawn(Params, From) ->
        F = crest_utils:get_lambda(Params),
    Key = crest_spawn:install(F),
        gen_server:reply(From, Key).

handle_exec({Key, Params}, From, Spawned) ->
        case dict:find(Key, Spawned) of
        {ok, ChildPid} ->
```

```
            Res = crest_utils:rpc(ChildPid, Params),
            log4erl:info("Executed the existing key ~p~
                n", [Key]),
                        gen_server:reply(From, {ok, Res
                            });
        error ->
                        gen_server:reply(From, {error})
    end.

handle_list(Param, From, Spawned) ->
        Result = dict:fold(fun(Key, Pid, AccIn) ->
                                Val = {Key, crest_utils:
                                    rpc(Pid, Param)},
                                [Val|AccIn]
                                end, [], Spawned),
    log4erl:info("Collected all responses for parameter
        ~p~n", [Param]),
        gen_server:reply(From, dict:from_list(Result)).
```

Listing B.2: CREST spawn handler

```
%% @author Alessandro Sivieri <alessandro.sivieri@mail.
    polimi.it>
%% @doc Process operations module; it is designed as a
    supervisor_bridge process.
%% It does not have the capability to restart a child.
%% @copyright 2010 Alessandro Sivieri

-module(crest_spawn).
-behaviour(supervisor_bridge).
-export([install/1, start/2, init/1, terminate/2]).

%% External API

%% @doc Install a new operation, specified by the
    function as parameter,
%% by spawning a new process associated to it (and
    linked to this bridge);
%% it returns the unique key associated to this
    function.
%% @spec install(fun()) -> string()
install(F) ->
    Key = uuid:to_string(uuid:srandom()),
    Params = {Key, {?MODULE, start, [Key, F]},
        temporary, infinity, supervisor, [?MODULE]},
    supervisor:start_child(crest_sup, Params),
    Key.

%% @doc Start the link of this bridge to the newly
    spawned function.
%% @spec start(string(), fun()) -> {ok, pid()}
start(Key, F) ->
    {ok, BridgePid} = supervisor_bridge:start_link(?
        MODULE, {Key, F}),
    {ok, BridgePid}.

%% @doc Add child pid and key to the crest_peer server
    here, for avoiding
%% problems with shutting down children.
```

```
%% @spec init({string(), fun()}) -> {ok, pid(), pid()}
init({Key, F}) ->
    ChildPid = proc_lib:spawn_link(fun() -> F() end),
    crest_peer:add_child(Key, ChildPid),
    {ok, ChildPid, ChildPid}.

%% @doc Terminate a child pid.
%% @spec terminate(any(), pid()) -> atom()
terminate(Reason, ChildPid) ->
    log4erl:info("Supervisor bridge ~p: terminating
       child ~p (~p)~n", [self(), ChildPid, Reason]),
    exit(ChildPid, Reason).
```

Listing B.3: Sample module

```
%% @author Someauthor <somemail@somedomain.xx>
%% @doc Sample module
%% @copyright yyyy Someauthor

-module(module_sample).
-export([get_function/0]).

%% External API

%% @doc Returns the closure to be executed remotely (as
    spawn or remote);
%% note that the function has to call itself tail-
    recursively, if it has
%% to be spawned (remote functions are invoked just
    once, and then their
%% process is ended).
%% It is required to correctly handle the param=name
    message, while the
%% other responses need to include the Content-Type
    parameter of the
%% answer data (see below).
%% How to send this closure to the CREST peer is left
    to the developer
%% (for example, through inets:http, or ibrowse).
%% @spec get_function() -> fun()
get_function() ->
    F = fun(F) ->
        receive
            {Pid, {"param", "name"}} ->
                Pid ! {self(), "Closure service name"},
                F(F);
            {Pid, {"param", "operation"}} ->
                Pid ! {self(), "GET/POST"},
                F(F);
            {Pid, {"param", "parameters"}} ->
                Pid ! {self(), [{"param1", "string()"},
                    {"param2", "integer()"}]},
                F(F);
```

```
            {Pid, [{Parameter1, Value1}, {Parameter2,
                Value2}]} ->
                computation(Value1, Value2),
                Pid ! {self(), {"text/plain",
                    ComputationResults}},
                F(F);
            {Pid, Other} ->
                Pid ! {self(), {"text/plain",
                    crest_utils:format("Error: ~p", [
                    Other])}},
                F(F)
    end
end,
fun() ->
    F(F)
end.
```

Appendice C

Esempio di trasmissione POST

```
POST /crest/spawn HTTP/1.1
content-type: application/x-www-form-urlencoded
te:
host: localhost
connection: keep-alive
Content-Length: 6982

module=spawn_test_1&
binary=FOR1%00%00%0BXBEAMAtom%00%00%00%DC%00
%00%00%17%0Cspawn_test_1%0Cget_function%04main%05inets%05start%04
http%0Bset_options%0Bcrest_utils%11get_lambda_params%04post%07
request%02ok%05error%06erlang%02%2B%2B%02io%06format%04halt%0B
module_info%0Fget_module_info%16-get_function%2F0-fun-1-%16
-get_function%2F0-fun-0-%04selfCode%00%00%01%F2%00%00%00%10%00%00
[...]&
filename=%2Fhome%2Falex%2FDocumenti%2FPolitecnico%2Fworkspace%2F
crest-demo%2Febin%2Fspawn_test_1.beam&
code=%83p%00%00%00%9B%00%14%D9F%8AM%12%C1G%97%C3%03%E4s%9C%7F%DC
%00%00%00%01%00%00%00%01d%00%0Cspawn_test_1a%01b%07g%0D%D5gd%00
%0Dnonode%40nohost%00%00%00%02%00%00%00%00%00p%00%00%00M%01%14
[...]
```