Let's Connect LuaTEX to the World

Roberto Giacomelli

Sommario

Una delle migliori novità del mondo TEX è LuaTEX non solamente per la tipografia in sé. Questo potente motore di composizione è capace di recuperare dati da molte sorgenti diverse, una caratteristica molto importante nei contesti di produzione di imprese e studi professionali, dove le informazioni condivise sono una risorsa vitale.

In questo articolo discuterò le soluzioni che ho trovato fino a ora durante la mia attività professionale, per estendere il sistema TEX con funzioni esterne come quelle dei driver SQL. Illustrerò poi l'ultima mia inattesa scoperta, sorprendentemente semplice e innovativa: *connettere il codice tramite messaggi*.

Niente più codice di basso livello di cui prendersi cura, niente più vincoli dovuti alle incompatibilità tra programmi, ma uno strato software fornito dalla libreria ZeroMQ con cui connettere LuaTEX a servizi esterni.

Illustrerò in dettaglio l'installazione di ZeroMQ, e con l'aiuto di esempi funzionanti, mostrerò come usufruire in LuaTEX di servizi costruiti con il linguaggio Rust.

Abstract

One of the best pieces of news in the TEX world is LuaTEX not only for typography itself. In fact, this very powerful typesetting engine is capable to retrieve data from a variety of sources, a very important feature especially in the production context such as business and professional teams, where shared data are a vital resource.

In this paper I will describe several solutions achieved so far during my job, to execute external code like SQL drivers and pass the datasets to TEX. Then, I will introduce the latest point of view that I suddenly discovered a few weeks ago, surprisingly simple and innovative as well: *connect code by messaging*.

No more low level artifacts to care about, no more constraints due to incompatibility between programs, but a communication layer provided by the ZeroMQ library connecting LuaTEX with distributed services.

I will detail how to install ZeroMQ and, with the help of several working examples, I will show how to benefit from services built with Rust within LuaTEX.

1 Introduzione

Per illustrare le tecnologie dei moderni strumenti del sistema TEX con cui è possibile realizzare documenti che fanno parte di processi produttivi, è utile definirne le principali caratteristiche e delineare alcuni punti critici che ne limitano la produzione.

Parliamo di relazioni tecniche, lettere, elenchi di oggetti, istruzioni, frutto del lavoro quotidiano di moltissime persone all'interno di aziende o studi professionali, la cui caratteristica essenziale è la correttezza dei dati che contengono.

In generale, le informazioni dell'organizzazione sono archiviate in qualche forma per poi essere elaborate e incluse all'interno di documenti. I modi in cui i dati vengono resi persistenti e poi utilizzati, ne determinano l'*affidabilità* e l'*efficienza*.

L'affidabilità è la misura della sicurezza di accesso ai dati e il mantenimento del loro stato di correttezza nel tempo, mentre l'efficienza è la misura della rapidità con cui vengono create, reperite e modificate le informazioni. Si tratta di fattori chiave poiché un buon flusso informativo abbassa i costi e incrementa evoluzione e miglioramento dei prodotti.

L'utilizzo di TEX non è quindi quello classico della scrittura di un singolo documento, come una tesi di laurea o un libro, ma la redazione di documenti di carattere aziendale dallo schema predefinito che includono dati condivisi nell'organizzazione.

Nella sezione 2 ripercorrerò i diversi metodi che ho sperimentato con lo scopo di migliorare l'affidabilità e l'efficienza dei dati, elaborando complesse collezioni di dati per generarne documenti con la qualità tipografica del sistema TEX.

Dalla sezione 3 svilupperò una nuova ulteriore idea basata sullo scambio di *messaggi* tra codici in esecuzione in processi indipendenti, introducendo ZeroMQ. Un'ampia unità applicativa incentrata sull'uso di servizi esterni in LuaTEX comprenderà la creazione di simboli QRCode e l'accesso a database attraverso la connessione a *socket* ZeroMQ secondo lo schema client/server.

La seconda parte dell'articolo tratterà delle questioni più tecniche: la sezione 8 riporterà i dettagli d'installazione e le principali risorse di documentazione sui software utilizzati. La sezione 9 sarà dedicata alle funzionalità FFI in LuaTEX e alla creazione di librerie dinamiche. Le tre sezioni successive tratteranno l'implementazione in Rust dei servizi server impiegati in precedenza.

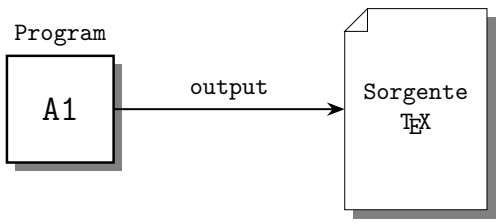


Figura 1: Schema di funzionamento della tecnologia di costruzione del sorgente TEX che permette l'indiretta e indipendente elaborazione esterna dei dati.

2 Far colloquiare il codice

Parte del problema più generale descritto nell'introduzione è dare una risposta alla seguente domanda: come implementare il codice che ricava i dati memorizzati all'interno di un archivio informatico e con essi comporre un documento con TEX?

Per riuscirci è essenziale estendere le funzioni di base del motore di composizione con moduli software specifici e il successo di queste tecnologie dipende da quanto sia sicuro far colloquiare questi moduli con TEX.

Le soluzioni che illustrerò con maggior dettaglio nelle prossime sezioni, sono:

- costruzione automatica del sorgente TEX;
- esecuzione di codice nativo Lua;
- esecuzione tramite Lua di codice compatibile C utilizzando l'interfaccia FFI (Foreign Function Interface);
- esecuzione tramite Lua di codice compatibile C utilizzando il binding diretto a librerie a caricamento dinamico;
- esecuzione tramite Lua di codice compatibile C utilizzando librerie a caricamento dinamico conformi all'API di Lua.

2.1 Costruzione del sorgente TEX

Il più semplice metodo d'elaborare i dati di un archivio è quello di scrivere un programma la cui uscita sia il file di testo contenente il codice TEX che una volta compilato dà origine al documento desiderato.

Questa tipo di tecnica è rappresentata nello schema di figura 1. Il programma `A1` si compone essenzialmente di due parti: la prima si occupa di predisporre i dati per la pubblicazione estraendoli dagli archivi, e la seconda di costruire con essi il file sorgente per uno dei formati della famiglia TEX.

Non vi è alcuna necessità di coinvolgere TEX in nessuna di queste due fasi perciò l'utente è completamente libero sia di scegliere il linguaggio di programmazione con cui implementarle, sia di scegliere il motore tipografico con cui sarà compilato il sorgente di output.

Il ruolo del sistema di composizione è secondario poiché l'attenzione è quasi sempre rivolta al codice che genera il sorgente, mentre dal punto di vista del flusso dati il file sorgente generato è in sostanza il modo con cui il programma `A1` *comunica* con TEX, ovvero con una forma statica e unilaterale memorizzata su disco.

Il vantaggio di questo metodo è la completa separazione tra il programma e il sistema TEX, tuttavia l'ulteriore indirezione aggiunta a quella sempre presente tra sorgente `.tex` e documento finale limita l'utente nell'apportare modifiche dovute a cambiamenti nei contenuti del documento o alla volontà di migliorarne l'aspetto.

Un'analisi della tecnica di generazione del sorgente TEX che comprende numerosi esempi, si può trovare nell'articolo (Giacomelli e Pignalberi, 2015) pubblicato su questa stessa rivista.

2.2 Esecuzione di codice esterno

Fino all'uscita del compositore LuaTEX non vi erano altre strade praticabili per raggiungere lo scopo che non rifarsi alla tecnica precedente di *costruzione del sorgente*.

LuaTEX apre numerosi e ampi scenari poiché, includendo Lua — un linguaggio di tipo dinamico implementato in C —, è in grado di eseguire codice in molti modi diversi. Diviene possibile, durante la compilazione di un sorgente `.tex`, eseguire un numero molto grande di librerie esistenti oppure codice ad alte prestazioni scritto appositamente, secondo lo schema di figura 2.

Tuttavia è fondamentale che l'esecuzione del codice esterno sia affidabile e sicura, caratteristica che però non è affatto scontata.

2.3 Foreign Function Interface

In generale si può definire la Foreign Function Interface — in breve FFI — una tecnologia che facilita a un programma scritto in un dato linguaggio di programmazione l'esecuzione di codice scritto in un altro linguaggio.

LuaTEX offre, o meglio offrirà, funzionalità stabili FFI nel prossimo futuro (si veda l'articolo

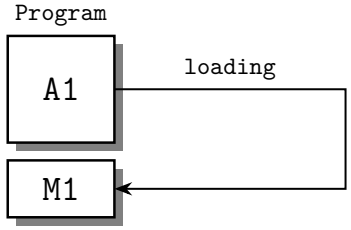


Figura 2: Schema generale di funzionamento della tecnologia di esecuzione di codice esterno. Nel programma principale `A1`, Lua carica il modulo `M1` che estende le funzionalità di base con quelle necessarie per l'elaborazione dei dati esterni. Una volta acquisite le informazioni, esse vengono composte nel documento da LuaTEX.

sull'argomento di Hans Hagen e Luigi Scarso (Hagen e Scarso, 2017)) mentre LuajitTEX già ne dispone perché FFI è incluso come estensione in LuaJIT, il compilatore Just In Time per Lua che sostituisce il classico interprete in questo motore di composizione relativamente poco noto agli utenti.

Grazie alla tecnologia FFI è possibile, per esempio, connettersi a database relazionali e comporre i risultati delle query direttamente in un sorgente TEX. L'idea può essere approfondita leggendo Giacomelli (2017), che spiega nel dettaglio l'installazione e la configurazione dei moduli necessari e fornisce esempi applicativi.

Un esempio d'uso di FFI è riportato alla sezione 9, con la compilazione di una libreria dinamica elementare scritta in Rust e l'esecuzione dell'unica funzione contenuta in essa all'interno di LuaTEX.

2.4 L'API di Lua

La seconda parte del libro di Roberto Ierusalimschy su Lua (Ierusalimschy, 2016) descrive l'API del linguaggio, ovvero come sia possibile aggiungere a esso nuove e potenti funzionalità attraverso lo stack virtuale e le relative funzioni in C.

Questo fa di Lua un linguaggio estensibile attraverso il caricamento dinamico di librerie binarie scritte appositamente. Alla sezione 9.2 vedremo un esempio di codice che ne chiarisce le modalità.

Questa terza via all'esecuzione di codice esterno è quella più intimamente legata al funzionamento interno di Lua, tutto incentrato sullo *stack*; questa struttura dati fu ritenuta la migliore soluzione dai progettisti del linguaggio per risolvere i conflitti tecnologici tra Lua e C, il linguaggio d'implementazione.

Grazie allo stack possiamo scambiare dati da e verso Lua dall'ambiente di esecuzione in C, per esempio scrivendo una funzione in C (o in Rust come nell'esempio concreto della sezione 9.2) che crea una tabella Lua. Chiaramente con questa tecnica potremmo scrivere in C un modulo per accedere a un database e renderne disponibili i dati all'interno di LuaTEX.

Nel farlo, dobbiamo occuparci di tutti i dettagli, definire i nomi delle funzioni dell'API di Lua che verranno risolti a runtime, lavorare con lo stack trovando la corrispondenza tra i tipi, quelli del C e quelli di Lua, scrivere la funzione di inizializzazione della libreria, e provare la validità del codice.

Possiamo definire l'operazione *un compito non banale dalle prestazioni eccellenti* sia in termini di tempi di esecuzione sia in termini di occupazione di memoria.

3 Canali di comunicazione

Mentre ragionavo sulle difficoltà tecniche di eseguire codice esterno in LuaTEX mi venne in mente una soluzione che avrebbe aggirato tutte le difficoltà senza rinunciare alla comunicazione bidirezionale tra i due ambienti di esecuzione: si trattava di sfruttare i canali dello standard input e output del sistema operativo come ponte tra due diversi programmi indipendenti.

In altre parole, il codice Lua eseguito in LuaTEX avrebbe avviato un comando esterno tramite la funzione `io.popen()` passando i dati da elaborare come argomenti. Il programma esterno avrebbe stampato i dati di uscita sul terminale così che, al termine della sua esecuzione, Lua avrebbe potuto acquisirli.

A dimostrazione pratica dell'idea, costruiamo il comando `double` con Rust[1] come un programma per il terminale che accetta un numero intero come primo e unico argomento e ne stampa il corrispondente valore doppio[2]:

```
// a simple Rust program to print the double
// of an integer
use std::env;
fn main() {
    let args: Vec<String> = env::args()
        .collect();
    let n: i32 = args[1]
        .parse()
        .unwrap();       // input
    println!("{}", 2*n); // output
}
```

Compiliamo il codice con il comando seguente lanciato in una sessione di terminale la cui directory di lavoro corrisponde a quella del file sorgente:

```
$ rustc double.rs
```

e verifichiamone il funzionamento con l'ulteriore comando:

```
$ ./double 34
68
```

Siamo ora pronti per passare a LuaTEX salvando il seguente sorgente nella stessa directory dell'eseguibile `double`:

```
% !TeX program = LuaTeX--shell-escape
\directlua{
local f = io.popen("./double 123", "r")
local res = f:read("*n")
f:close()
tex.print(res)
}
\bye
```

Aggiungendo l'opzione `--shell-escape` al comando di compilazione per permettere l'esecuzione della funzione `io.popen()`[3], LuaTEX dovrebbe produrre un file PDF contenente il numero 246.

1. La sezione 8.2 è dedicata alla presentazione di questo linguaggio.
2. Per semplicità il codice dell'esempio non gestisce gli eventuali errori dovuti alla non validità dell'argomento.
3. La funzione della libreria interna di Lua `io.popen()` accetta un comando esterno, apre un file virtuale come suo canale di output e ne restituisce un riferimento.

L'idea è molto semplice ma consente la comunicazione bidirezionale tra Lua, che richiede un servizio, e il componente esterno, che esegue quanto richiesto restituendo al mittente il risultato.

3.1 L'entrata in scena di ZeroMQ

Diverso tempo fa io e Luigi Scarso avemmo in chat una conversazione il cui argomento era l'affidabilità in scrittura dei dati su un database SQLite3. Luigi mi propose l'idea di costruire un server attraverso ZeroMQ — una libreria open source che non conoscevo — e far colloquiare diversi processi d'esecuzione attraverso messaggi usando protocolli riconosciuti.

Il server avrebbe reso più sicure le operazioni di scrittura e quindi aumentato l'affidabilità del database. In quel momento utilizzavo per l'operazione degli script in LuaJIT per popolare un database con i dati di centinaia di impianti elettrici condominiali inseriti in un programma di manutenzione; tali dati comprendevano voci di opere, computi metrici, stati d'avanzamento lavori, ecc.

Grazie a quella conversazione, il passaggio dall'idea di usare i canali di input/output del sistema operativo a quella di usare ZeroMQ è stato quasi immediato: una volta ottenuto l'accesso a ZeroMQ, LuaTEX avrebbe potuto richiedere dati ed elaborazioni a programmi esterni in modo sicuro e affidabile, semplice e veloce.

4 Lo schema Client/Server

I due programmi della sezione precedente, LuaTEX da un lato e `double` dall'altro, comunicano con il *message pattern* chiamato Client/Server.

Descrivendo a parole lo schema riportato nella figura 3, durante la compilazione del sorgente LuaTEX, il codice Lua `A1` invia un messaggio che rappresenta l'informazione da elaborare a un preciso *indirizzo* ed entra poi nello stato di attesa sullo stesso canale.

Il programma *server* `A2`, in ascolto a quell'indirizzo, riceve il messaggio, compie le elaborazioni necessarie e inoltra al processo client la risposta. Di conseguenza, il programma client si riattiva per elaborare i dati contenuti nel messaggio di risposta.

Da questa semplice descrizione, possiamo subito ricavare le seguenti osservazioni:

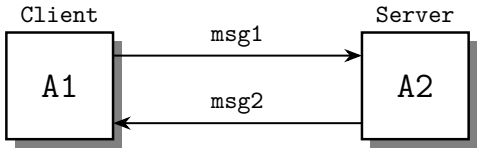


Figura 3: Schema di comunicazione Client/Server tra il programma `A1` che invia un messaggio al programma `A2` e quest'ultimo che risponde reinviando verso il mittente un secondo messaggio.

- il server deve essere attivo e disponibile;
- sia il client che il server devono conoscere l'indirizzo esatto di scambio dei messaggi;
- sia il client che il server dovranno conoscere le specifiche di formato dei messaggi perché sia possibile la loro interpretazione;
- un messaggio è una sequenza di byte;
- di per sé lo scambio dei messaggi non identifica il tipo di nodo, in particolare se client o server; è invece l'ordine con cui essi sono spediti a farlo. Più in generale, è il *comportamento* dei nodi a definirne la dinamica delle relazioni. Il sistema di comunicazione è unicamente tale, non influisce cioé sul comportamento dei nodi.

4.1 Il doppio di 28

Riprendendo l'esempio illustrato all'inizio della sezione precedente con cui un numero veniva raddoppiato, implementiamo la tipologia a due nodi costruendo un server in Rust attravero il progetto `rust-zmq` ospitato alla pagina `https://github.com/erickt/rust-zmq` e un client in Lua con la libreria `lzmq` in esecuzione all'interno di un sorgente LuaTEX[4].

Il server, ricevuto un testo che rappresenta un intero, ne calcola il doppio e reinvia questo valore al mittente. Il client invia il testo che rappresenta un numero intero al server e prosegue l'esecuzione una volta acquisita la risposta.

La libreria ZeroMQ è progettata per essere intuitiva da usare: nel codice, il reinvio della risposta da parte del server non necessita dell'individuazione del client ma semplicemente dell'invio di un messaggio al *socket*. Il server inoltre risponde a qualsiasi client evadendo le richieste in ordine di arrivo e, ancora una volta, ci pensa ZeroMQ a gestire questa coda.

4.1.1 Preparazione del progetto Rust

Come primo passo creiamo con il package manager `cargo` un progetto Rust aprendo un terminale sulla directory scelta per ospitare i file:

```
$ cd ~
$ mkdir project
$ cd project
$ cargo new basic_server --bin
$ cd basic_server
$ tree .
.
|-- Cargo.toml
+-- src
    +-- main.rs

1 directory, 2 files
```

4. L'installazione e le configurazioni dei moduli necessari per compilare ed eseguire il codice proposto, sono illustrate nella sezione 8 dell'articolo.

Editiamo poi il *manifest* `Cargo.toml` appena creato in automatico modificando il nome del pacchetto in `startserver`, e dichiarando la dipendenza *zmq* alla versione 0.8[5]:

```
[package]
name = "startserver"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
zmq = "0.8"
```

4.1.2 Il server

Editiamo adesso il file `main.rs` nella cartella `src` del progetto, in modo che contenga il seguente codice:

```
extern crate zmq;
fn main() {
    let ctx = zmq::Context::new();
    let responder = ctx.socket(zmq::REP)
                       .unwrap();
    assert!(
        responder.bind("tcp://*:5555")
                 .is_ok()
    );
    let mut msg = zmq::Message::new()
                               .unwrap();
    println!("Server ready...");
    loop {
        responder.recv(&mut msg, 0);
        // message decoding
        let n: i32 = msg.as_str()
                        .unwrap()
                        .parse()
                        .unwrap();
        println!("Received [{}]", n);
        // reply
        responder.send_str(
            &(2 * n).to_string()[..], 0
        ).unwrap();
    }
}
```

Anche se sono scritti in Rust i passaggi hanno validità generale:

1. per prima cosa viene creato l'oggetto di tipo `Context` che rappresenta il processo nel quale è in esecuzione ZeroMQ, affidandone il riferimento alla variabile `ctx`;

2. subito dopo otteniamo un socket predisposto per il REPLAY dei messaggi e infatti al metodo `socket()` del contesto si passa la costante predefinita `zmq::REP`;

3. colleghiamo ora al socket un indirizzo di comunicazione per il protocollo `TCP` situato alla porta numero 5555 dell'host locale, che altri non è che lo stesso computer su cui stiamo lavorando, ma potremmo indicare un qualsiasi IP sulla rete locale;

4. creiamo adesso un oggetto di tipo `Message` in cui depositare il contenuto delle richieste. Nella libreria `rust-zmq` esistono funzioni di più alto livello per ottenere dal socket i messaggi ma quell'oggetto può essere riutilizzato più volte essendo in effetti un contenitore e ciò rende il server ancor più veloce.

A questo punto la connessione è stabilita e il server è pronto per rispondere ai messaggi. Potremmo considerare di rispondere a un'unica richiesta ma conviene prevedere un ciclo in modo da lasciare il server sempre attivo, e poter compilare tutte le volte che si vuole il sorgente LuaTEX che ogni volta richiederà i servizi. Sarà tuttavia necessario interromperne l'esecuzione del server con la combinazione di tasti CTRL+C:

1. all'interno del ciclo infinito mettiamo in ricezione il server chiamando il metodo `recv()` del socket con il puntatore all'oggetto di tipo `Message` e un flag di valore 0, che richiede che l'operazione di ricezione sia in *blocking mode* così da attendere l'arrivo delle richieste;

2. alla linea di codice successiva il messaggio sarà contenuto nell'oggetto `msg`. Una catena di chiamate, molto comuni in Rust, ne esegue il metodo `as_str()` che restituisce il messaggio in forma di stringa. A sua volta su questo dato si esegue il metodo `parse()`[6] da cui si ricava l'intero che ho previsto sia a 32 bit con segno (tipo `i32`);

3. non rimane che inviare al socket un messaggio che giungerà nel formato testo al richiedente, chiamandone il metodo `send_str()` a cui si affida la stringa che rappresenta il valore doppio dell'intero e il flag 0 che regola il modo blocking o non-blocking dell'invio, qui ininfluente.

Si noti che non è necessario chiamare i metodi `close()` dell'oggetto messaggio, `disconnect()` del socket e `destroy()` del contesto, perché ciò avviene automaticamente grazie all'implementazione interna della libreria `rust-zmq` — quegli oggetti infatti implementano tutti il trait `Drop` — e alla gestione della memoria di Rust.

5. Potremo anche non esplicitare la versione della dipendenza — che segue le specifiche *semver* — indicando un asterisco al posto del numero di release. Così facendo sarà l'ultima versione disponibile di quella libreria a essere scaricata e utilizzata ma potremmo incorrere nella mancata risoluzione della versione nei casi con schemi complessi di dipendenze ramificate e circolari. Per questo motivo specificare un numero di versione è la norma.

6. Il metodo `parse()` sfrutta due altre funzionalità molto potenti di Rust: l'*inferenza*, quella capacità del compilatore di determinare il tipo degli oggetti senza che lo sviluppatore debba farlo esplicitamente, e il meccanismo dei *trait*, un modo per far sì che i tipi aderiscano a un'interfaccia„ cosa che tra l'altro rende generiche le funzioni.

Per scrivere il codice Rust che ho appena illustrato è necessario tenere a portata di mano la documentazione di `rust-zmq` e conoscere il tipo generico `Result<T, E>` definito nella libreria standard di Rust e brevemente introdotto alla sezione 8.2.1. È infatti molto importante essere consapevole dell'effetto dell'inserimento nel codice dei metodi `unwrap()`.

4.1.3 Compilazione del server

Per scaricare le dipendenze, compilare ed effettuare i necessari link alle librerie dinamiche è sufficiente dare il seguente comando in una finestra di terminale che punti alla directory principale del progetto, quella cioé che contiene il *manifest* `Cargo.toml`:

```
$ cargo build
```

Il package manager fa un gran lavoro ma la procedura non andrà a buon fine se non avremo installato i file di sviluppo di ZeroMQ come illustrato alla sezione 8.3.

4.1.4 Il client

Il client è un sorgente LuaTEX — ed è quindi scritto per il formato plain — che salveremo con il nome di `client.tex` in una directory per esempio `~/Scrivania/test_client`:

```
% !TeX program = LuaTeX--shell-escape
\input luapackageloader.sty
\directlua{
local zmq = require "lzmq"

print("Connecting to the server...")
local context = zmq.init(1)
local socket = context:socket(zmq.REQ)
socket:connect("tcp://localhost:5555")
print("Connection ready...")

local n = 28
print("Sending [" .. n .. "] to the server")
socket:send(tostring(n))

local reply = socket:recv()
socket:close()
context:term()
tex.print("Received... [" .. reply .. "]")
}
\bye
```

Il sorgente si apre caricando il pacchetto luapackageloader che modifica i percorsi di ricerca di script e librerie LuaTEX allargandone il campo anche al di fuori di TEX Live. Infatti, il binding a ZeroMQ `lzmq`, installato via Luarocks secondo le istruzioni della sezione 8.3, si trova nella directory inclusa nel percorso di ricerca dei normali script Lua, ovvero in `/usr/local/lib/lua/5.2/`. Questo significa che potremmo raggiungere il server anche scrivendo un normale script in Lua.

Il codice è analogo a quello del server con la differenza che basta l'invio e la ricezione di un solo messaggio. In sequenza si tratta di:

1. ottenere un riferimento al contesto con il metodo `init()` e da questo un riferimento a un oggetto socket attraverso il metodo `socket`, specificando la costante `REQUEST` che ne caratterizza il tipo, scelto in base al message pattern corrispondente;
2. stabilire la connessione con il metodo `connect` del socket all'indirizzo di protocollo `TCP` identico a quello del server. La comunicazione avverrà tra processi in esecuzione sullo stesso computer;
3. inviare al socket il messaggio di testo che rappresenta il numero 28, tramite il metodo `send()`;
4. ricevere il messaggio di testo di ritorno del server, che rappresenta il numero 56, tramite il metodo `recv()` del socket;
5. chiudere sia il socket con il metodo `close()`, sia il contesto con il metodo `term()`.

Terminata la comunicazione tra client e server, il codice prosegue con un'ultima istruzione eseguendo tramite la funzione `tex.print()` l'invio del messaggio di ritorno verso TEX che comporrà sulla pagina il testo seguente:

```
Received... [56]
```

4.1.5 Esecuzione

È il momento di provare se tutto funziona avviando il server in una finestra di terminale con il comando[7]:

```
$ cd ~/project/basic_server/target/debug
$ ./startserver
```

e compilando il nostro sorgente LuaTEX in una seconda finestra di terminale:

```
$ cd ~/Scrivania/test_client
$ luatex --shell-escape client.tex
```

5 Multipart messaging

In ZeroMQ un messaggio è una sequenza di uno o più *frame*, ciascuno dei quali composto dallla lunghezza in byte del contenuto seguito dal contenuto stesso, come illustrato nella figura 4. ZeroMQ garantisce che i messaggi multiparte siano spediti effettivamente in un unico blocco solo quando viene inviato l'ultimo frammento.

7. L'eseguibile del server, dopo la compilazione per mezzo del package manager `cargo`, si trova nella sotto-cartella `target/debug` della directory del progetto in Rust. La versione ottimizzata e priva delle informazioni di debug si ottiene invece aggiungendo al comando `cargo build` l'opzione `--release` e l'eseguibile si troverà nella sotto-cartella `target/release`. Si può anche compilare e lanciare il programma con il seguente unico comando `$ cargo run`, con o senza il flag di ottimizzazione.

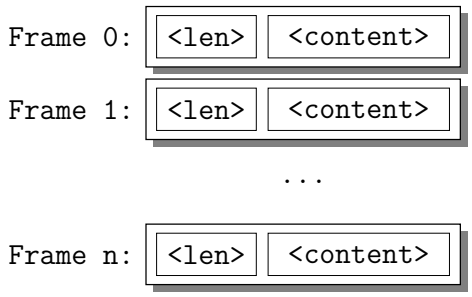


FIGURA 4: Costruzione del messaggio in ZeroMQ: l'insieme di uno o più frame ciscuno dei quali composto dalla lunghezza della sequenza di byte del contenuto e la sequenza stessa.

La composizione di un messaggio in più frame è il modo più semplice per strutturare l'informazione da trasmettere. Per esempio, il contenuto del primo frame potrebbe determinare se il messaggio di richiesta è stato elaborato con successo oppure se si sono verificati degli errori.

In questa sezione impiegheremo i messaggi multiparte affinché si possa includere il QRCode di un testo in un sorgente LuaTEX: il testo dell'indirizzo web della pagina del forum del sito GUIT dei post più recenti `http://www.guitex.org/home/it/forum/recent-topics`.

Per far sì che l'attenzione si focalizzi sul codice LuaTEX, la spiegazione dell'implementazione del server è rimandata più avanti alla sezione 10. Dal lato TEX dividerò il problema in quattro parti:

1. definizione del formato dei messaggi multiparte;
2. messa a punto del codice Lua per l'invio e la ricezione dei messaggi;
3. messa a punto del codice Lua per il disegno di una griglia generica;
4. verifica di funzionamento, con la scrittura del sorgente LuaTEX che conterrà il simbolo QRCode con il link previsto.

FIGURA 5: Il simbolo QRCode della pagina del sito GUIT dei post più recenti del forum, ottenuto in LuaTEX interrogando un server esterno, come illustrato nel testo.

5.1 QRCode

Il QRCode *Quick Response Code* (`https://it.wikipedia.org/wiki/Codice_QR`) si presenta come una griglia regolare $n \times n$ di piccoli elementi, a loro volta quadrati, che possono essere di colore nero oppure bianco. La figura 5 mostra l'esempio di simbolo QRCode che ci proponiamo di realizzare, verificabile scandendo il simbolo con uno smartphone.

Per prima cosa definiamo il formato del messaggio di richiesta: useremo molto semplicemente tre frame[8] dalla struttura seguente:

```
Frame 0: "QRCODE"
Frame 1: <encoding message>
Frame 2: "END"
```

Il servizio esterno potrà rispondere con due diversi messaggi identificati dal contenuto del frame iniziale `"QRCODE"` o `"ERR"`, per rappresentare il successo o meno dell'operazione di codifica del QRCode. Sulla base del frame 0 il successivo frame dovrà essere interpretato rispettivamente come la codifica del simbolo oppure come la descrizione dell'errore:

```
Frame 0 : "QRCODE"
Frame 1 : <qrcode 0/1 sequence>
Frame 2 : "END"
```

oppure:

```
Frame 0 : "ERR"
Frame 1 : <error description>
Frame 2 : "END"
```

5.2 Connessione al servizio

La connessione a una porta di un socket in ZeroMQ può dar luogo a errori. Un modo semplice ed efficace per gestire eventuali problemi è scrivere le funzioni Lua in modo che restituiscano due valori: in caso di successo, in prima posizione ci sarà il valore atteso e in seconda il valore `nil`, mentre all'opposto in caso di errore, in prima posizione ci sarà il valore `nil` e in seconda la stringa contenente la descrizione dell'errore stesso.

Nel file `libmsg.lua` cominciamo quindi con il creare una tabella in cui memorizzeremo le funzioni con il paradigma degli oggetti, e con lo scrivere la funzione `service()` che dalla stringa che descrive la porta del servizio, tenterà di crearne il socket:

```
-- libmsg.lua file
local libmsg = {}
libmsg.__index = libmsg

local zmq = require "lzmq"

function libmsg:service(port)--> table, err
    -- Connecting to the server
    local ctx, err = zmq.context{io_thread=1}
    if err then
        return
```

8. Come richiesto dal server esterno il contenuto testuale dovrà essere conforme alle specifiche Unicode UTF-8.

```
        nil, "[Context Error] "..err:msg()
    end
    local skt, err = ctx:socket(zmq.REQ)
    if err then
        return
        nil, "[Socket Error] "..err:msg()
    end
    local _, err = skt:connect(port)
    if err then
        return
        nil, "[Socket Error] "..err:msg()
    end

    local o = {
        context = ctx,
        socket  = skt,
    }
    setmetatable(o, self)
    return o, nil
end
```

La connessione al socket si potrà poi dismettere con la seguente funzione `disconnect()` che in caso di errore ne restituisce la descrizione:

```
function libmsg:disconnect() --> nil or error
    local _, err = self.socket:close()
    if err then
        return "[Socket error] "..err:msg()
    end
    local _, err = self.context:term()
    if err then
        return "[Context error] "..err:msg()
    end
end
```

Le ultime linee contengono la definizione della funzione centrale che ho chiamato `send_and_recv()`, quella che invia un messaggio multiparte e restituisce la risposta, e l'istruzione `return` che ritorna la tabella di libreria quando il file sarà richiamato da codice Lua con la funzione `require()`.

Il binding per Lua di ZeroMQ `lzmq` semplifica molto il lavoro con i messaggi multiparte grazie alla funzione `Socket:send_all()` a cui si passa l'array contenente la collezione ordinata dei frame:

```
function libmsg:send_and_recv(tmsg)--> t, err
    local skt = self.socket
    -- multipart message
    local _, err = skt:send_all(tmsg)
    if err then
        return
        nil, "[Socket Error] "..err:msg()
    end

    local dataset, err = skt:recv_all()
    if err then
        return
        nil, "[Socket Error] "..err:msg()
    end
    return dataset
end

return libmsg
-- end of libmsg.lua file
```

5.3 Disegno del simbolo

Dal servizio remoto riceveremo in risposta alla stringa di testo da codificare in un simbolo QR-Code, la sequenza di zero e uno raggruppabili in righe dall'alto verso il basso grazie al carattere di separazione `\n`. Associando al numero 1 il significato di quadratino pieno e al numero 0 quello di quadratino vuoto, la sequenza può essere disegnata in modo vettoriale sfruttando i nodi *pdfliteral* di LuaTEX, come mostrato nel dettaglio nell'articolo (Giacomelli, 2016) apparso su Ars 22.

Allo scopo, creiamo un nuovo file `libgrid.lua` con le stesse modalità del precedente, che come prima cosa definisca la funzione costruttore di un oggetto griglia a partire dalla stringa di zeri e di uno. Nella funzione `new()` ho considerato anche il carattere di separazione spazio perché nelle prove con sorgenti LuaTEX il segno di ritorno a capo è convertito automaticamente in un carattere spazio da TEX durante l'espansione dei token.

Nel dettaglio, la funzione elabora la sequenza in ingresso trasformandola in un array di valori booleani — `true` corrisponderà al quadratino da disegnare e `false` a nessuna azione — e negli interi contenenti il numero di righe e di colonne, perciò la griglia può anche essere rettangolare. Per gestire casi di errore, i dati di uscita saranno ancora la coppia di ⟨*valore*⟩/⟨*errore*⟩:

```
-- libgrid.lua file
local libgrid = {}
libgrid.__index = libgrid

-- constructor
function libgrid:new(qrseq) --> obj, err
    local seq = {}
    local row = 1
    for c in string.gmatch(qrseq, '.') do
        if c == '1' then
            seq[#seq + 1] = true
        elseif c == '0' then
            seq[#seq + 1] = false
        elseif c == '\n' or c == ' ' then
            row = row + 1
        else
            return
            nil, "Unexpected char '"..c.."'"
        end
    end
    local col = math.floor(#seq/row)
    if not (col*row == #seq) then
     return
     nil,
     "The sequence doesn't fit the rectangle"
    end
    local o = {
        qrseq = seq,
        row = row,
        col = col,
    }
    setmetatable(o, self)
    return o
end
```

Senza entrare troppo nei dettagli che si trovano nell'articolo citato, i codici in notazione postfissa previsti dal formato PDF per disegnare i rettangoli sono i seguenti:

```
q
<x> <y> <width> <height> re
<x> <y> <width> <height> re
...
f
S
Q
```

che genereremo con la funzione `_pdfliteral()`, da intendersi privata. La funzione crea questo testo calcolando le coordinate x e y dell'angolo inferiore sinistro di ciascun quadratino a partire dall'ultima riga in basso e in direzione da destra a sinistra, e inserendo la dimensione del *modulo* in punti grandi (*bp*), per la misura dei lati:

```
function libgrid:_pdfliteral(mod_bp)
    mod_bp = mod_bp or 4 -- module size
    local seq = self.qrseq
    local row, col = self.row, self.col

    local pdflit = {"q"}
    local fmtpair = "%0.6f %0.6f"
    local fmtdim = string.format(fmtpair,
        mod_bp, mod_bp
    )
    local fmt = string.format("%s %s re",
        fmtpair, fmtdim
    )
    -- fmt => <x> <y> <width> <height> re
    local x, y = 0.0, 0.0
    for r = row-1, 0, -1 do
        local start = r*col
        for c = 1, col do
            if seq[start + c] then
                pdflit[#pdflit + 1] =
                    string.format(fmt, x, y)
            end
            x = x + mod_bp
        end
        x = 0.0
        y = y + mod_bp
    end
    pdflit[#pdflit + 1] = "f"
    pdflit[#pdflit + 1] = "S"
    pdflit[#pdflit + 1] = "Q"
    return table.concat(pdflit, "\n"), mod_bp
end
```

L'ultima funzione necessaria è quella che costruisce una scatola orizzontale contenente il disegno della griglia. Non sarebbe difficile scriverla se non fosse che i nodi di tipo *pdfliteral* hanno dimensione nulla perciò, se non adeguassimo il contenuto della scatola, il disegno si sovrapporebbe al testo o supererebbe i margini della pagina.

Non solo. Dobbiamo anche prevedere una cornice bianca attorno al simbolo spessa almeno quattro volte la dimensione del modulo, detta *quiet zone*, per consentire la scansione del simbolo del codice a barre da parte dei dispositivi.

La funzione `boxpack()` si occuperà della creazione della scatola orizzontale con le corrette dimensioni. Essa inizia con il controllare l'esistenza del registro box il cui nome è contenuto nella variabile `boxreg`, corrispondente a quello che avremo avuto l'accortezza di definire in precedenza nel sorgente con il classico comando primitivo TEX `\newbox`. Una volta ottenuto il codice *pdfliteral* con l'omonima funzione, la costruzione della scatola avviene con la concatenazione di nodi di tipo diverso, seguendo attentamente i seguenti passi[9]:

passo a) costruzione del nodo *pdfliteral* `n0`;

passo b) costruzione del nodo spazio elastico `vqz0` di altezza pari alla quiet zone;

passo c) concatenazione dei nodi `n0` e `vqz0`. Il nodo `n1` restituito dalla funzione di concatenazione è un puntatore allo stesso nodo `n0`;

passo d) costruzione della scatola verticale `vbox1` con il nodo di testa `n1`;

passo e) costruzione della distanza elastica orizzontale `hqz1` di larghezza pari alla quiet zone;

passo f) creazione lista `n2` con nodo distanza orizzontale e nodo scatola verticale;

passo g) inserimento finale della lista nella scatola orizzontale `hbox2` a cui poi si impongono le dimensioni in larghezza e altezza della griglia compreso la cornice di quiet zone.

Traducendo le istruzioni in codice Lua otteniamo la funzione:

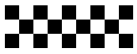
```
function libgrid:boxpack(boxreg, mod_bp)
    assert(
        tex.isbox(boxreg),
        string.format(
           "Box register [%s] doesn't exist",
           boxreg
        )
    )
    local pdf, mod = self:_pdfliteral(mod_bp)
    -- symbol dimensions (scaled point)
    mod = mod * tex.sp "1bp"
    local width  = self.col * mod -- sp
    local height = self.row * mod -- sp
    local quietzone = 4 * mod     -- sp

    -- step a: pdfliteral node
    local n0 = node.new(
        "whatsit", "pdf_literal"
```

9. Per una maggiore comprensione, ho numerato con lo stesso indice le variabili dei nodi che sono allo stesso livello nella lista e assegnato un passo diverso ogni volta che viene creato un nuovo nodo.

```
    )
    n0.data = pdf

    -- step b: vertical quietzone dim
    -- (respect to the baseline)
    local vqz0 = node.new "glue"
    vqz0.width = quietzone

    -- step c: vlist
    local n1 = node.insert_after(
        n0, n0, vqz0
    )
    -- step d: vertical box
    local vbox1 = node.vpack(n1)

    -- step e: horizontal dim
    local hqz1 = node.new "glue"
    hqz1.width = quietzone
    -- step f: left horizontal quietzone
    local n2 = node.insert_after(
        hqz1, hqz1, vbox1
    )
    -- step g: final horizontal box
    local hbox2 = node.hpack(n2)
    hbox2.width  = width + 2 * quietzone
    hbox2.height = height + 2 * quietzone
    tex.box[boxreg] = hbox2
end

return libgrid
-- end of libgrid.lua file
```

5.3.1 *Verifica di* `libgrid`

La griglia rettangolare di prova:



è correttamente disegnata dal codice:

```
% !TeX program = LuaTeX
\nopagenumbers
\newbox\mybox
\directlua{
local libgrid = require "libgrid"
local grid, err = libgrid:new
    [[101010101
      010101010
      101010101]]
assert(not err, err)
grid:boxpack "mybox"
}
\leavevmode\box\mybox
\bye
```

5.4 Il sorgente LuaTEX conclusivo

Il passo finale è la realizzazione del QRCode con il link che punta alla pagina dei post recenti sul forum del GUIT, simbolo riportato nella figura 5. Nel sorgente LuaTEX riportato qui sotto una buona parte del codice Lua è impegnato nelle operazioni di parsing del messaggio di risposta:

```
% !TeX program = LuaTeX--shell-escape
\input luapackageloader.sty
\nopagenumbers
\newbox\mybox

\directlua{
local libmsg = require "libmsg"
local libgrid = require "libgrid"
local QRCodeSrv, err = libmsg
      :service "tcp://localhost:5556"
assert(not err, err)

local dataset, err = QRCodeSrv:send_and_recv{
   "QRCODE",
   "http://www.guitex.org/home/"
   .."it/forum/recent-topics",
   "END"
}
assert(not err, err)

local err = QRCodeSrv:disconnect()
assert(not err, err)

local cmd    = dataset[1]
local msg    = dataset[2]
local endmsg = dataset[3]
assert(endmsg == "END")

if cmd == "ERR" then
    error(msg)
end

local grid, err = libgrid:new(msg)
assert(not err, err)

local width = grid.row
assert(grid.row == grid.col)

grid:boxpack "mybox"
}
\leavevmode\box\mybox
\bye
```

6 Un server, più servizi

I messaggi non sono vincolati a una struttura fissa. Ne deriva che un servizio remoto può incorporare più di un tipo di elaborazione. Per esempio, il client potrebbe inviare al server un messaggio multiparte il cui primo frame è il codice che corrisponde a una particolare elaborazione.

In questa sezione utilizzeremo questo dato per realizzare un servizio remoto multifunzione, con cui richiedere tre differenti elaborazioni sui numeri primi:

`PRIME-CHECK:` test di primalità;

`PRIME-COUNT:` conteggio dei primi in un intervallo;

`PRIME-LIST:` lista dei primi in un intervallo.

Per poterci concentrare sul client in LuaTEX, rimando alla sezione 11 la costruzione del server in Rust.

6.1 Messaggi e funzioni

Il formato generale dei messaggi di richiesta prevede più frame e, in sostanza, è equivalente alla chiamata di una funzione che ne specifica il nome e gli argomenti di ingresso con questa struttura:

```
Frame 0: <function ID>
Frame 1: <arg1>
Frame 2: <arg2>
...
Frame n: "END"
```

Altrettanto generale può essere il formato dei messaggi di risposta:

```
Frame 0: <function ID>
Frame 1: <result1>
Frame 2: <result2>
...
Frame n: "END"
```

6.2 Funzione di alto livello

L'idea è di scrivere la funzione `libmsg:call()` che accetta il nome della funzione remota disponibile sul server, e un numero variabile di argomenti che dipende dal servizio stesso. Questa funzione di alto livello dovrà eseguire i seguenti passi:

1. codifica del messaggio di richiesta;
2. invio del messaggio di richiesta e ricezione della risposta;
3. decodifica del messaggio di risposta.

Aggiungiamo alla libreria in Lua `libmsg`, costruita nella sezione precedente, due funzioni per la codifica e la decodifica dei messaggi che restituiscono una tabella Lua contenente i frame del messaggi:

```
-- generic encoding of the request message
function libmsg:encode(fname, ...)
    local tmsg = {fname}
    for _, elem in ipairs {...} do
        tmsg[#tmsg + 1] = tostring(elem)
    end
    tmsg[#tmsg + 1] = "END"
    return tmsg
end

-- generic decoding of the replay message
function libmsg:decode(tmsg)
    if tmsg[1] == "ERR" then
        assert(tmsg[3] == "END")
        return nil, tmsg[2]
    end
    assert(tmsg[#tmsg] == "END")
    return tmsg
end
```

La funzione `call()` è la diretta implementazione dei tre passi dell'elenco di apertura di sezione. Essa restituisce i valori alternativamente validi della tabella Lua con i risultati in prima posizione oppure della descrizione dell'errore in seconda posizione:

```
function libmsg:call(fname, ...)
    local tmsg = self:encode(fname, ...)
    local res, err = self:send_and_recv(tmsg)
    if err then
        return nil, err
    end
    return self:decode(res)
end
```

6.3 Verifica di primalità

Per come è implementato il server, i risultati di tutte le funzioni remote sono sequenze di interi, perciò il servizio `PRIME-CHECK` ritorna la lista contenente un solo valore, 0 se il numero trasmesso nel messaggio di richiesta è primo, altrimenti 1. Qui di seguito il listato di un sorgente LuaTEX in cui la verifica di primalità è ottenuta semplicemente chiamando la funzione Lua `Service:call()`:

```
% !TeX program = LuaTeX--shell-escape
\input luapackageloader.sty

\directlua{
local libmsg = require "libmsg"
PrimeSrv = libmsg
           :service "tcp://localhost:5557"
}
\def\checkprime#1{\directlua{
local n = tonumber(#1)
local ans, err = PrimeSrv
                 :call("PRIME-CHECK", n)
assert(not err, err)
local isprime = tonumber(ans[2])
assert(isprime == 0 or isprime == 1)

if isprime == 1 then
    tex.print("prime")
else
    tex.print("not prime")
end
}}

The number 62643217 is \checkprime{62643217}
while 920419813 is \checkprime{920419813}.

\directlua{
local err = PrimeSrv:disconnect()
if err then error(err) end
}
\bye
```

6.4 Tabella dei primi

Il risultato del secondo esempio, riferito al servizio remoto che restituisce la lista dei primi in un dato intervallo, è riportato in tabella 1. Il contenuto del messaggio di risposta può quindi essere piuttosto grande: esso infatti dedica un frame per ciascuno dei primi della lista. Tuttavia nelle prove non ho notato alcun problema né alcun evidente rallentamento di ZeroMQ.

Il sorgente da compilare con LuaLATEX è il seguente. Una volta ottenuta dal servizio remoto la lista dei primi, la creazione della tabella avviene stampando

Tabella 1: Tabella dei primi inferiori a 1000 ottenuta in LuaLaTeX scambiando messaggi con un server per mezzo della libreria ZeroMQ, come spiegato in dettaglio nel testo. Questa tabella è ispirata a quella identica che compare nelle prime pagine del libro "L'Ossessione dei numeri primi" di John Derbyshire.

2	3	5	7	11	13	17	19	23	29	31	37	41	43
47	53	59	61	67	71	73	79	83	89	97	101	103	107
109	113	127	131	137	139	149	151	157	163	167	173	179	181
191	193	197	199	211	223	227	229	233	239	241	251	257	263
269	271	277	281	283	293	307	311	313	317	331	337	347	349
353	359	367	373	379	383	389	397	401	409	419	421	431	433
439	443	449	457	461	463	467	479	487	491	499	503	509	521
523	541	547	557	563	569	571	577	587	593	599	601	607	613
617	619	631	641	643	647	653	659	661	673	677	683	691	701
709	719	727	733	739	743	751	757	761	769	773	787	797	809
811	821	823	827	829	839	853	857	859	863	877	881	883	887
907	911	919	929	937	941	947	953	967	971	977	983	991	997

il corpo dell'ambiente tabular con la funzione interna di LuaTeX `tex.print()` come se fosse stato digitato manualmente, ma con il numero di colonne specificate nella definizione della macro `\column`:

```
% !TeX program = LuaLaTeX--shell-escape
\documentclass[margin=2pt]{standalone}
\usepackage{luatex85}
\usepackage{luapackageloader}
\usepackage{lmodern}
\usepackage{booktabs}

\directlua{
    local libmsg = require "libmsg"
    PrimeSrv = libmsg
        :service "tcp://localhost:5557"
}
\def\column{14}

\begin{document}
\catcode`\#=12
\begin{tabular}{*{\column}{c}}
\toprule
\directlua{
local list, err = PrimeSrv:call(
    "PRIME-LIST", 1, 1000
)
assert(not err, err)

local len = #list
local col = \column
local row = math.floor(len/col)
if col * row < len then
    row = row + 1
end
local dbs = string.char(92, 92)
for r = 1, row do
    local i = (r-1) * col + 1
    local j; if r == row then
        j = len
    else
        j = i + col - 1
    end
    tex.print(
        table.concat(list, "&", i, j)..dbs
    )
end
}\bottomrule
\end{tabular}
\directlua{
local err = PrimeSrv:disconnect()
assert(not err, err)
}
\end{document}
```

6.5 Conteggio dei primi

Il grafico di figura 6 è stato ottenuto grazie alla *funzione* `PRIME-COUNT` disponibile sul server. Nel sorgente che lo produce, riportato di seguito, le coordinate dei punti sono inserite nel flusso dei token nel momento in cui viene espansa la primitiva `\directlua` che si trova all'interno dell'argomento della macro `\addplot`.

Il server viene interrogato ogni volta che occorre conoscere il numero dei primi nell'intervallo di estremi $[a, a+s)$ con s costante e a che si incrementa ogni volta di s. La costruzione del grafico di figura 6 richiede in particolare 281 richieste al server, e anche qui non si apprezza nessun rallentamento durante la compilazione.

```
% !TeX program = LuaLaTeX--shell-escape
\documentclass[margin=2pt]{standalone}
\usepackage{luatex85}
\usepackage{luapackageloader}
\usepackage{lmodern}
\usepackage{pgfplots}
\pgfplotsset{compat=1.15}

\footnotesize
\directlua{
    local libmsg = require "libmsg"
    PrimeSrv = libmsg
        :service "tcp://localhost:5557"
}
\def\nstep{100}   % interval size
\def\npoints{281} % number of points

\begin{document}
\begin{tikzpicture}
```

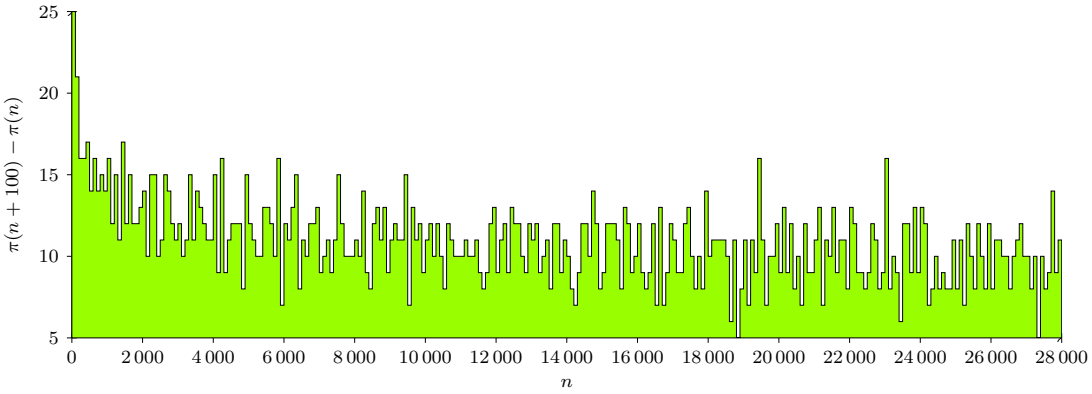


FIGURA 6: Grafico in pgfplots risultato della compilazione del sorgente LuaLaTeX riportato nel testo in cui si ricava il conteggio dei numeri primi per ogni intervallo consecutivo di ampiezza 100, da un servizio esterno raggiunto tramite la libreria ZeroMQ.

```
\begin{axis}[
const plot,
width=16.5cm, height=6.5cm,
scaled ticks=false,
ylabel = $\pi(n+\nstep) - \pi(n)$,
xlabel = $n$,
axis x line=bottom,
axis y line=left,
axis line style={->},
tick style={color=black},
tick label style={%
    /pgf/number format/1000 sep={\,}},
    font=\footnotesize
]
\addplot[
    black,
    fill=green!40!yellow
] coordinates {
    \directlua{
    local step, points = \nstep, \npoints
    local a = 0
    local perc = string.char(37)
    local fmt = "("..perc.."d,"..perc.."s)"
    for _ = 1, points do
        local tprime, err = PrimeSrv:call(
            "PRIME-COUNT", a, a + step - 1
        )
        assert(not err, err)
        local p = tprime[1]
        tex.print(
            string.format(fmt, a, p)
        )
        a = a + step
    end
    }
}
\closedcycle;
\end{axis}
\end{tikzpicture}
\directlua{
local err = PrimeSrv:disconnect()
assert(not err, err)
}
\end{document}
```

7 Database server

Il tema di questa quarta e ultima sezione applicativa dell'articolo è l'accesso a un database — una delle più importanti funzionalità richieste in ambito aziendale — tramite la tecnologia dello scambio di messaggi.

Dal punto di vista del sorgente LuaTeX — ovvero il nodo che assume il ruolo di client — non solo non si conosce come è stato implementato il server, ma nemmeno quale sia il reale motore SQL del database. Le uniche informazioni necessarie sono solamente:

1. l'indirizzo del server,
2. il formato dei messaggi.

7.1 Messaggi e query SQL

Il server accetta messaggi di richiesta che contengono al secondo frame il testo della query SQL e in quelli successivi gli eventuali parametri corrispondenti. Possiamo quindi effettuare ogni sorta di interrogazione nei confronti del database connesso con il server a patto di conoscerne la struttura.

Poiché ogni database implementa il linguaggio SQL con alcune varianti rispetto alla versione standard, dovremmo in realtà conoscere il motore effettivo dell'archivio SQL per poter interagire con più efficacia con i dati. Il server impiegherà in effetti SQLite3 http://www.sqlite.org/, una delle soluzioni più semplici e affidabili per la gestione dati oggi disponibili.

Per incrementare la sicurezza dei dati, il server non accetterà le query in scrittura che modificano il database. Questa condizione corrisponde allo scopo del client, che è quello di reperire le informazioni per comporre documenti, senza alcuna necessità di apportare modifiche.

In conclusione, poiché il messaggio di richiesta contiene il testo della query SQL, dobbiamo anche conoscere:

3. la struttura del database,

4. le specifiche esatte del linguaggio SQL.

7.2 Formato dei messaggi

La sequenza dei frame del messaggio di richiesta è la seguente:

```
Frame 0: "QUERY"
Frame 1: <prepared statement SQL>
Frame 2: <param1>
Frame 3: <param2>
...
Frame n: "END"
```

I parametri dal frame 2 in poi sono richiesti solo nel caso in cui il *prepared statement* li preveda. Eseguire le interrogazioni tramite questa funzione rende più sicuri i dati, impedendo azioni malevole di *SQL injection* quando i parametri sono definiti all'esterno. Nel caso del server di questa applicazione, il database rimane comunque in sola lettura e la definizione dei parametri è fatta dall'utente TEX.

Se ci sono stati errori o azioni non ammesse il server risponde con il seguente messaggio multiparte:

```
Frame 0: "ERR"
Frame 1: <error description>
Frame 2: "END"
```

altrimenti invia un messaggio piuttosto complesso, articolato in tre sezioni: la prima contiene il numero c delle colonne della tabella risultato della query, la seconda è la lista dei nomi delle colonne e la terza è la lista dei dati della tabella di r righe, una dopo l'altra, ed è quindi lunga $c \cdot r$:

```
Frame 0    : "TABLE"
Frame 1    : <column counter c>
Frame 2    : <column name 1>
Frame 3    : <column name 2>
...
Frame 2 + c: <column name c>
Frame 3 + c: <data row 1 col 1>
Frame 4 + c: <data row 1 col 2>
...
Frame 2 + (r+1)*c: <data row r col c>
Frame 3 + (r+1)*c: "END"
```

Come verificheremo nel prossimo esempio, il server aggiunge delle chiavi a ciascun elemento della tabella per caratterizzarne il tipo. SQLite infatti ammette che i dati possano essere di un tipo diverso rispetto a quello dichiarato per il campo della tabella, una scelta simile a quella dei linguaggi dinamici come Lua e Python nei confronti dei tipi dei dati. Il controllo dei tipi, invece, permetterebbe di non appesantire i dati associando il tipo alla colonna corrispondente.

L'elenco dei prefissi è il seguente:

`NULL:` tipo nullo in SQL;

`INTE:` intero con segno a 64 bit;

`REAL:` numero in virgola mobile a 64 bit;

`TEXT:` stringa di testo;

`BLOB:` stream di byte[10].

7.3 Verifica di base

Una volta avviato il servizio, eseguiamo una semplice query per ottenere la data impostata in SQLite. La libreria Lua `libmsg` è la stessa dei progetti precedenti e il relativo file deve essere presente nella cartella del seguente sorgente LuaTEX:

```
% !TeX program = LuaTeX--shell-escape
\input luapackageloader.sty
\directlua{
local libmsg = require "libmsg"
DBSrv, err = libmsg
    :service "tcp://localhost:5558"
assert(not err, err)
}

\directlua{
local qmsg, err = DBSrv:call(
    "QUERY", [[SELECT DATE() AS date;]]
)
assert(not err, err)
tex.print(
    table.concat(
        qmsg, string.char(92)..[[par ]]
    )
)
}

% closing the service's connection
\directlua{
local err = DBSrv:disconnect()
assert(not err, err)
}
\bye
```

Il PDF, risultato della compilazione con l'opzione `--shell-escape` di attivazione dei comandi di sistema, dovrebbe contenere un testo simile al seguente, con l'intera struttura dei frame del messaggio di risposta:

```
TABLE
1
date
TEXT:2017-08-11
END
```

7.4 Iteratori di tabelle

Creiamo un nuovo file `libiter.lua` con le stesse modalità dei precedenti: conterrà codice che semplifichi l'iterazione delle righe di una tabella contenuta nel messaggio di risposta del server come risultato di una query.

Per esempio, con questo iteratore potremo scrivere il codice di richiesta (query) dei dati contenuti nella tabella `departments` del database di prova

10. `BLOB` sta per Binary large object.

Tabella 2: Semplice elenco dei dipartimenti contenuti nel database di esempio ottenuto con LuaTEX attraverso un servizio remoto e un iteratore scritto appositamente e spiegato nel testo.

d001 — Marketing
d002 — Finance
d003 — Human Resources
d004 — Production
d005 — Development
d006 — Quality Management
d007 — Sales
d008 — Research
d009 — Customer Service

(illustrato in dettaglio nella prossima sezione) e formanti l'elenco riportato nella tabella 2:

```
% !TeX program = LuaTeX--shell-escape
\input luapackageloader.sty
\nopagenumbers
\directlua{
libiter = require "libiter"
local libmsg = require "libmsg"
DBSrv, err = libmsg
    :service "tcp://localhost:5558"
assert(not err, err)
}

\directlua{
local qmsg, err = DBSrv:call(
    "QUERY",
    [[SELECT dept_no, dept_name
      FROM departments;]]
); assert(not err, err)

for n, row in libiter.rows(qmsg) do
    tex.print(
        row.dept_no, "---", row.dept_name,
        string.char(92)..[[par ]]
    )
end
}

% closing the service's connection
\directlua{
local err = DBSrv:disconnect()
assert(not err, err)
}
\bye
```

Nella parte centrale del sorgente è presente l'iteratore (funzione `libiter.rows()`) in azione all'interno di un normale ciclo `for` di Lua. L'iteratore accetta direttamente la tabella contenente il messaggio multiparte con i dati restituiti dalla query. A ogni iterazione abbiamo a disposizione una prima variabile contenente il numero progressivo del record — non utilizzata nell'esempio — e un secondo oggetto indicizzabile con i nomi delle colonne.

L'iteratore è tecnicamente uno *stateless iterator* che impiega una metatabella per consentire l'indicizzazione riga per riga dei valori corrispondenti alle colonne della query. È il genere di cose che rende Lua un linguaggio elegante.

Il seguente è il codice contenuto nel file di libreria `libiter.lua`:

```
local libiter = {}
local transform = {
    ["NULL:"] = function (_) return nil end,
    ["INTE:"] = function (x)
        return tonumber(x)
    end,
    ["REAL:"] = function (x)
        return tonumber(x)
    end,
    ["TEXT:"] = function (x) return x end,
    ["BLOB:"] = function (_)
        return "unimplemented!"
    end,
}

local function convert(s)
    local key = s:sub(1,5)
    assert(transform[key])
    return (transform[key])(s:sub(6))
end

-- stateless iterator
function libiter.rows(tmsg)
    local cols = tonumber(tmsg[2])
    local colpos = {}
    local k = 1
    for i = 3, 2 + cols do
        colpos[tmsg[i]] = k
        k = k + 1
    end

    local obj = {}
    local mt = {}
    setmetatable(obj, mt)

    function mt.__index(t, key)
        if not colpos[key] then
            error(string.format(
                "The column name '%s' "..
                "doesn't exist",
                key
            ))
        end
        local pos = colpos[key]
        local nrow = t[0]
        local idx = 2 + nrow * cols + pos
        local res = tmsg[idx]
        return convert(res)
    end
    local function iter(o, nrow)
        nrow = nrow + 1
        o[0] = nrow
        local idx = 3 + nrow * cols
        if tmsg[idx] ~= "END" then
            return nrow, o
        end
    end
    return iter, obj, 0
end
return libiter
```

7.5 Il database di prova

Per effettuare le prove ho adattato il database del progetto https://github.com/vrajmohan/pgsql-sample-data a SQLite riducendolo a sole tre tabelle riguardanti la definizione dei dipartimenti, che abbiamo già incontrato alla sezione precedente, e l'anagrafica degli impiegati di un'ipotetica azienda.

Il codice SQL che genera le tabelle su cui il server eseguirà le query è il seguente. Esso rappresenta l'intera struttura del database di prova.

```
CREATE TABLE departments (
    dept_no     CHAR(4) PRIMARY KEY,
    dept_name   VARCHAR(40) NOT NULL,
    UNIQUE (dept_name)
);

CREATE TABLE employees (
    emp_no      INTEGER PRIMARY KEY,
    birth_date  CHAR(10)    NOT NULL,
    first_name  VARCHAR(14) NOT NULL,
    last_name   VARCHAR(16) NOT NULL,
    gender      CHAR(1)     NULL,
    hire_date   CHAR(10)    NOT NULL
);

CREATE TABLE dept_emp (
    emp_no      INT      NOT NULL,
    dept_no     CHAR(4)  NOT NULL,
    from_date   DATE     NOT NULL,
    to_date     DATE     NOT NULL,
    FOREIGN KEY (emp_no)
    REFERENCES employees (emp_no)
    ON DELETE CASCADE,
    FOREIGN KEY (dept_no)
    REFERENCES departments (dept_no)
    ON DELETE CASCADE,
    PRIMARY KEY (emp_no, dept_no)
);
```

7.6 Badge degli impiegati

L'esempio riportato in figura 7 contiene due *badge* dimostrativi di altrettanti impiegati memorizzati nel database. A sinistra compare il simbolo QRCode corrispondente all'identificativo personale mentre a destra una colonna dati con, tra le altre informazioni, il nome e il dipartimento di appartenenza del dipendente. In una situazione reale non sarebbero comprese le date personali per il rispetto della privacy.

A realizzare i badge è il seguente sorgente LuaLaTeX:

```
% !TeX program = LuaLaTeX--shell-escape
\documentclass[margin=2pt]{standalone}
\usepackage{luatex85}
\usepackage{luapackageloader}
\usepackage{lmodern}
\usepackage{booktabs}

% build the services
\directlua{
libgrid = require "libgrid"
```

ID: 10001
Name: Georgi Facello
Birth Date: 1953-09-02
Dept.: Development
Hire date: 1986-06-26
From: 1986-06-26

ID: 10002
Name: Bezalel Simmel
Birth Date: 1964-06-02
Dept.: Sales
Hire date: 1985-11-21
From: 1996-08-03

Figura 7: Due badge dimostrativi realizzati con LuaLaTeX come illustrato nel testo, impiegando la libreria ZeroMQ per accedere a due servizi esterni, il primo per ricavare il QRCode dell'identificativo dell'impiegato e il secondo per interrogare il database contenente l'anagrafica.

```
libiter = require "libiter"
local libmsg = require "libmsg"
DBSrv, err = libmsg
    :service "tcp://localhost:5558"
assert(not err, err)

QRCodeSrv, err = libmsg
    :service "tcp://localhost:5556"
assert(not err, err)
}

\begingroup
\catcode`\#=12
% execute the query against the employees db
\directlua{
local employees, err = DBSrv:call(
"QUERY", [[SELECT
employees.emp_no      AS empid,
employees.first_name  AS firstname,
employees.last_name   AS lastname,
employees.birth_date  AS birthdate,
employees.hire_date   AS hiredate,
departments.dept_name AS dept,
dept_emp.from_date    AS fromdate
FROM departments, dept_emp, employees
WHERE
departments.dept_no = dept_emp.dept_no AND
employees.emp_no = dept_emp.emp_no AND
dept_emp.to_date = '9999-01-01'
ORDER BY empid
LIMIT ?;]], 2
); assert(not err, err)
Employees = employees

function printmacro(macro, ...)
    local s = {string.char(92)}
    s[#s+1] = macro
    for _, a in ipairs({...}) do
```

```
        s[#s+1] = "{"..a.."}"
    end
    tex.print(table.concat(s))
end
function printqrcode(empid)
    local tmsg, err = QRCodeSrv
        :call("QRCODE", empid)
    assert(not err, err)
    local grid = libgrid:new(tmsg[2])
    tex.box["mybox"] = grid:boxnode()
    printmacro("box")
    printmacro("mybox")
end
}
\endgroup

\newbox\mybox

% macro to typeset each row
\def\printqrcode#1{%
\begin{minipage}[c]{40mm}
\directlua{printqrcode(#1)}
\end{minipage}}
%
\def\minicell#1#2#3#4#5#6{%
\begin{tabular}{@{}l@{}}
#1\\#2\\#3\\#4\\#5\\#6
\end{tabular}}

% typesetting the table
\begin{document}
\begin{tabular}{cl}
\toprule
\directlua{
local isfirst = true
for _, emp in libiter.rows(Employees) do
    if isfirst then
        isfirst = false
    else
        printmacro("midrule")
    end
    printmacro("printqrcode", emp.empid)
    tex.print("&")
    printmacro("minicell",
    "ID: "        ..emp.empid,
    "Name: "      ..emp.firstname
                  .." "..emp.lastname,
    "Birth Date: "..emp.birthdate,
    "Dept.: "     ..emp.dept,
    "Hire date: " ..emp.hiredate,
    "From: "      ..emp.fromdate
    )
    tex.print(string.char(92, 92))
end
}
\bottomrule
\end{tabular}
% closing services' connections
\directlua{
local err = DBSrv:disconnect()
assert(not err, err)
local err = QRCodeSrv:disconnect()
assert(not err, err)
}
\end{document}
```

Dopo aver effettuato le connessioni ai due servizi per il QRCode e per l'accesso al database degli impiegati, come primo passo si esegue la query su tutte e tre le tabelle dell'archivio per reperire i dati.

Nella colonna `to_date` della tabella `dept_emp`, la data `'9999-01-01'` indica che l'impiegato in questione è in servizio presso il dipartimento, perciò nella query imporremo tale valore.

La query contiene come parametro il numero delle righe che si desidera vengano al massimo incluse nella tabella risultato. Questo valore — pari a 2 nell'esempio — verrà utilizzato dal database al momento dell'esecuzione della query come prepared statement. Ciò dimostra la possibilità di creare una macro TEX che tra gli argomenti accetti parametri per le interrogazioni verso il database.

Se non vi sono errori, il messaggio di risposta dal server in lettura sul database è contenuto nella variabile globale `Employees` perché possa essere iterata in seguito con la funzione `libiter.rows()`.

All'interno dell'ambiente `document` compare un ambiente `tabular` a due colonne: nella prima verrà inserito il QRCode e nella seconda i dati anagrafici per mezzo di due corrispondenti macro: `\printqrcode` e `\minicell`. Esse *stampano* i caratteri con la funzione `tex.print()` che poi TEX elaborerà.

Questa tecnica è molto semplice; tuttavia, nello scrivere il codice, occorre tener presente che tutti i caratteri stampati cominceranno a essere elaborati solo alla conclusione del codice Lua della macro `\directlua` che costruisce la tabella. Non funzionerebbe quindi generare la scatola orizzontale con il QRCode e stampare i caratteri `\box\mybox` per riempire la prima cella perché TEX giunto alla fase di espansione troverà l'ultimo QRCode nel primo `hbox` della prima riga della tabella ed espanderà una scatola ormai vuota nelle successive.

Una soluzione è quella di stampare nella prima cella una macro seguita dal codice da tradurre in QRCOde, per esempio `\printqrcode{10001}`. Questa macro conterrà il codice Lua per reperire dal servizio illustrato alla sezione 5 la stringa di zeri e uno che poi la libreria `libgrid` tradurrà in una scatola orizzontale con il contenuto grafico del simbolo.

L'espansione della `\directlua` sarà vuota ma TEX si ritroverà i seguenti caratteri al termine dell'iterazione sui record reperiti dal database, a meno degli spazi e delle interruzioni di riga inseriti per maggiore chiarezza:

```
\printqrcode{10001} & \minicell{10001}
{Georgi Facello}{1953-09-02}{Development}
{1986-06-26}{1986-06-26}
\midrule
\printqrcode{10002} & \minicell{10002}
{Bezalel Simmel}{1964-06-02}{Sales}
{1985-11-21}{1996-08-03}
```

La funzione Lua `printmacro()` accetta infatti un primo argomento come nome della macro che stampa nello streaming del compositore dopo avervi premesso il carattere di backslash, e un numero variabile di argomenti che stampa uno dopo l'altro in sequenza dopo averli inseriti tra graffe.

Da notare che non funzionerebbe la soluzione di usare la funzione `node.write()` di LuaTEX per riempire la prima cella di ciascuna riga della tabella dei badge perché avrebbe l'effetto di inserire immediatamente la scatola orizzontale contenente il QRCode nella lista principale del compositore, e quindi inserirebbe i codici a barre tutti nella prima cella della prima riga.

Usare la tecnologia LuaTEX dei nodi per realizzare interamente la tabella darebbe ovviamente il risultato corretto ma esula dagli obiettivi dell'articolo che non sono incentrati sul problema di come far fluire i dati da Lua verso il compositore.

8 Risorse

In questa sezione citerò le risorse, sia software sia di documentazione, per la comprensione e l'esecuzione del codice riportato in questo articolo. Sono richieste una preparazione di base sulla programmazione e la familiarità con la riga di comando della shell.

Nonostante i software impiegati, tutti rigorosamente open source, siano disponibili per diversi sistemi operativi, tutte le analisi, le prove e le compilazioni sono state effettuate in ambiente Linux, in particolare per la distribuzione Xubuntu basata su Debian. I dettagli d'installazione potranno quindi essere differenti per i sistemi operativi diversi da quello del pinguino.

Prepariamo gli ingredienti per questa nostra particolare ricetta che sono, nell'ordine, il motore di composizione di nuova generazione LuaTEX, il linguaggio Rust e la libreria ZeroMQ.

8.1 LuaTEX

Il riferimento principale per LuaTEX, in particolare per la versione 1.0.4 distribuita con la TEX Live 2017, è il suo manuale curato direttamente dal team di sviluppo (The LuaTEX development team, 2017) e accessibile da terminale digitando il comando usuale:

```
$ texdoc luatex
```

Nel manuale non si trovano ancora informazioni sulla FFI — Foreign Function Interface, si veda in proposito l'articolo (Hagen e Scarso, 2017) — poiché, a differenza di LuajitTEX, in questo motore di composizione esse sono state aggiunte di recente a livello sperimentale. Un utile riferimento si trova sul sito del progetto LuaJIT `http://luajit.org/ext_ffi_tutorial.html`, poiché l'estensione dovrebbe esporre la stessa API sia per LuaTEX che per LuajitTEX.

Per l'installazione di TEX Live 2017, e quindi di LuaTEX, conviene scegliere una delle modalità previste e specificate alla pagina `https://www.tug.org/texlive/`. Se si dispone già della distribuzione 2017 o successiva, raccomando di aggiornare TEXLive con il comando da adattare in base al percorso dell'eseguibile `tlmgr`:

```
$ sudo /path/to/tlmgr update --all
```

8.2 Rust

Il C sarebbe dovuto essere uno dei protagonisti di questo lavoro, tuttavia non sono in confidenza con questo linguaggio. D'altra parte Rust, il relativamente nuovo linguaggio open source patrocinato da Mozilla Foundation, ha come obiettivo principale la sicurezza e la stabilità dei programmi senza che i controlli ne appesantiscano l'esecuzione. Inoltre Rust è in grado di interoperare da e verso il C facilmente e può produrre file binari per librerie statiche o dinamiche, o anche file oggetto.

Rust nasce con un sistema di gestione della memoria progettato per eliminare i problemi di consistenza e sicurezza dei dati tipici del C come i casi di puntatori a oggetti non più esistenti — i *dangling pointer* —, di memoria allocata e non più utile — il *memory leak* — o l'uso di variabili non inizializzate. Tutti questi casi sono intercettati in fase di compilazione e non comportano un aggravio dei tempi di elaborazione durante l'esecuzione del codice, mentre il superamento degli indici di un array provoca l'uscita del programma.

8.2.1 Uno sguardo a Rust

In Rust ci sono molti nuovi concetti che derivano dal complesso di studi ed esperienze fatti sia nell'area dei linguaggi in stile C, sia nell'area dei linguaggi funzionali come Haskell e OCalm. In questa sezione cercherò di illustrare in modo semplificato la caratteristica principale di questo linguaggio progettato per la programmazione di sistema e del suo compilatore chiamato `rustc` che produce codice macchina nativo e che ha tra i suoi componenti fondamentali LLVM al pari del linguaggio Clang utilizzato per gli stessi scopi sulle piattaforme OSX.

Come è naturale attendersi, le differenze tra Rust e Lua, il linguaggio incorporato in LuaTEX, sono profonde: in Rust occorre preoccuparsi di *come* i dati vengono memorizzati e di *quali* diritti si dispone per elaborarli. Il beneficio che si ottiene è un codice molto più veloce e sicuro, adatto per progetti come componenti di sistemi operativi, driver e moduli di basso livello. Uno dei principali progetti sviluppati in Rust è Servo, il browser web di Mozilla che dovrebbe nel prossimo futuro sostituire Firefox. Tuttavia la sintassi moderna e altre caratteristiche come l'inferenza dei tipi avvicinano Rust alla produttività dei linguaggi dinamici come Lua e Python.

In Rust la chiave del controllo statico della memoria è la semantica dell'*ownership*. Il linguaggio

chiarisce e rende espliciti i concetti della rappresentazione in memoria dei dati al prezzo di una curva di apprendimento più ripida e un maggiore impegno nella progettazione. Nel modello di memoria di Rust vi sono due modi di memorizzare un valore:

unboxed value dato nello *stack*; la memoria è liberata automaticamente non appena cessa la validità dello stack, ovvero al termine dello *scopo* per cui è stato definito;

owned boxed dato nell'*heap*, la memoria dinamica indipendente dai blocchi di scopo delle variabili, e creazione di un riferimento ad esso nello stack *unico proprietario* del dato; la memoria è liberata automaticamente quando ha termine lo scopo in cui è definito il riferimento, tenendo conto dei *passaggi di proprietà*.

I dati unboxed sono oggetti la cui vita è regolata dal blocco in cui vengono creati. Nel momento in cui questo contesto termina di esistere, anche gli oggetti che contiene vengono eliminati. Un *box* invece fa riferimento a un'area nell'*heap*: i valori in questa memoria meno efficiente dello stack, assumono una vita indipendente dal contesto. Nasce il problema di come rendere sicuro l'uso di questi dati e in particolare di come distruggerli.

In C l'operazione è manuale! Nel codice occorre fare estrema attenzione alla validità dei puntatori e all'eliminazione volontaria dei dati. All'opposto in Lua l'operazione è automatica. Un componente software, chiamato *Garbage Collector*, durante l'esecuzione del programma controlla quali oggetti non sono più raggiungibili ovvero non esiste più alcun riferimento che li indirizzi. Per scelta di progetto, gli errori di memoria in Rust sono intercettati a tempo di compilazione e ciò rende in pratica inutile un Garbage Collector.

La soluzione adottata in Rust è far sì che il boxed value, di tipo `T`, sia di *proprietà* del riferimento, di tipo `Box<T>`: quando il riferimento esce di scopo viene eseguito il corrispondente distruttore che libera correttamente la memoria indirizzata. In gioco ci sono quindi due oggetti: il primo è il boxed value che occupa un'area della memoria dinamica dell'heap e il secondo è il suo riferimento "*owner*" che si trova memorizzato nella memoria di lavoro dello stack e di cui il compilatore è in grado di conoscerne la validità in fase di compilazione.

La proprietà di un boxed value può cambiare grazie a quella che viene chiamata *move semantic*. Questo consente di prolungare la vita di un dato dell'heap oltre a un singolo constesto di scopo.

Il prossimo esempio evidenzia come non sia possibile violare la regola di unicità del riferimento proprietario di un boxed value. Se copiamo il riferimento in uno nuovo — la variabile `a` nella `b` nel codice — quello precedente non può più essere utilizzato. La proprietà passa al nuovo riferimento:

```
let mut a: Box<i32> = Box::new(100);
let b = a;
*a += 100; // <- compiler error:
// use of moved value: `*a`
```

così come non è possibile istanziare più di un puntatore mutevole, ovvero un riferimento in grado di modificare l'oggetto indirizzato:

```
let mut a: Box<i32> = Box::new(100);
let p1 = &mut a;
let p2 = &mut a; // <- compiler error:
// cannot borrow `a` as mutable more than
// once at a time
```

oppure ancora non è possibile modificare un valore di cui già esiste un puntatore mutevole:

```
let mut a: Box<i32> = Box::new(100);
let p1 = &mut a;
*a += 1; // <- compiler error:
// cannot assign to `*a` because it is borrowed
```

Ne consegue che una funzione non può creare valori boxed e restituirne solamente un puntatore:

```
fn crea(n: i32) -> &'static i32 {
    let mut b: Box<i32> = Box::new(n);
    &b // <- compiler error:
    // `b` does not live long enough
}

fn main() {
    let a = crea(100);
}
```

dovrà invece restituire il riferimento boxed la cui proprietà passerà alla variabile nel contesto della chiamata della funzione:

```
fn crea(n: i32) -> Box<i32> {
    Box::new(n) // OK
}

fn main() {
    let b = crea(100); // 'b' is the new owner
    let p = &b;
}
```

Altra caratteristica di Rust utile da conoscere è la non esistenza del tipo nullo, fonte di problemi nei programmi in C. Viene sostituito dal tipo generico `Option<T>` implementato nella libreria standard come `enum`, tipo che ammette che si possa associare dati alle varianti. La particolare enumerazione `Option<T>` consta di due varianti: `Some<T>` e `None`, la prima per il caso di esistenza del dato e la seconda per il caso contrario.

Nella libreria standard esiste anche un'altra enumerazione che evita l'uso del tipo nullo, il tipo generico `Result<T, E>` definito nella libreria standard come:

```
enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

con la prima variante si rappresenta un risultato corretto di tipo `T`, mentre con la seconda si rappresenta un errore di tipo `E`.

Per esempio, una funzione che restituisce il rapporto tra due interi potrebbe essere definita così[11]:

```
fn divide(n: i32, d: i32)
    -> Result<i32, &'static str> {
    if d == 0 {
        Err("Divisor is equal to zero")
    } else {
        Ok(n/d)
    }
}

fn main() {
    println!("{}", divide(518, 0).unwrap())
}
```

Il metodo `unwrap()` del tipo `Result` restituisce il valore associato alla variante `Ok` oppure produce l'interruzione del programma a runtime.

8.2.2 Risorse su Rust

La documentazione su Rust è già abbastanza nutrita anche se il primo riferimento disponibile, chiamato *the book* e sviluppato con il modello open source (The Rust Community, 2017), è attualmente in fase di completa riscrittura. Tra breve ne uscirà anche una versione a stampa (Klabnik e Nichols, 2017), come pure è quasi terminato il testo Blandy e Orendorff (2016).

8.2.3 Installare Rust

Per installare Rust su Linux — in questo lavoro ho utilizzato la versione 1.19 stabile — di solito si scarica lo script `rustup-init.sh` dal sito ufficiale `https://rustup.rs/` e lo si esegue oppure, più brevemente, si lancia il seguente unico comando da terminale:

```
$ curl https://sh.rustup.rs -sSf | sh
```

Se di Rust avete già un'installazione allora aggiornatela all'ultima disponibile, sapendo che l'avanzamento di versione avviene con un ciclo semestrale, con il comando:

```
$ rustup update stable
```

Una volta completate le procedure automatiche d'installazione, oltre al compilatore `rustc`, si hanno a diposizione `cargo`, il gestore dei pacchetti e dei task di compilazione, `rustdoc` per la generazione della documentazione e `rustfmt`, un'utilità che formatta il codice secondo uno standard. È possibile leggere il *libro* della guida con il comando:

```
$ rustup doc --book
```

mentre la documentazione completa della libreria standard si apre altrettanto comodamente nel browser con il comando:

```
$ rustup doc --std
```

8.3 ZeroMQ

E veniamo all'ultimo dei nostri ingredienti, ZeroMQ, una libreria scritta in C che semplifica lo scambio di messaggi tra due programmi attraverso un protocollo comune. Questa comunicazione può avvenire con diversi tipi di schemi a seconda delle necessità, il più semplice dei quali è lo schema[12] client/server denominato REQUEST/REPLY: un programma server in ascolto a un certo indirizzo, risponde alle richieste dei client, che possono essere sorgenti LuaTEX in fase di compilazione.

Nelle prossime due sezioni vedremo come documentarsi e come installare i moduli necessari per usare ZeroMQ, questa incredibile libreria che è stata adottata anche dal CERN di Ginevra per creare alcuni componenti chiave di gestione degli esperimenti sugli acceleratori di particelle (Dworak *et al.*, 2011).

8.3.1 Risorse su ZeroMQ

Il riferimento principale che troviamo in diverse forme è la guida chiamata `zguide` e scritta da Pieter Hintjens, uno dei principali sviluppatori, purtroppo scomparso l'anno scorso. La guida si può leggere online alla pagina `http://zguide.zeromq.org/page:all`, oppure nel libro scaricabile nel formato PDF (Hintjens, 2013a). Ne esiste anche un libro pubblicato per i tipi di O'Reilly (Hintjens, 2013b).

8.3.2 Installare ZeroMQ

La libreria ZeroMQ è scritta in linguaggio C; molti progetti, tuttavia, mettono a disposizione il *binding*[13] per molti altri linguaggi e piattaforme compreso Lua.

La procedura che ho messo a punto su un sistema Linux Debian-based comincia con l'installazione dei compilatori C e C++, dei file di sviluppo di Lua 5.2, e di quelli di ZeroMQ, nel momento in cui scrivo alla versione 4.1.4, utilizzando i repository ufficiali della distribuzione:

```
$ sudo apt-get install build-essential
$ sudo apt-get install liblua5.2-dev
$ sudo apt-get install libzmq3-dev
```

Se uno di questi pacchetti fosse già installato sul vostro sistema, non c'è da preoccuparsi, il comando corrispondente non avrà effetto. Si deve invece fare attenzione alla versione del pacchetto di sviluppo

11. Il costrutto `if` in Rust è un'espressione, restituisce cioè un risultato. Il vincolo è che tutti i rami del costrutto restituiscano lo *stesso* tipo, cosa verificata al solito dal compilatore.

12. Questi schemi vengono chiamati *communication pattern* o *message pattern*.

13. In generale per *binding* s'intende un componente software in grado di accedere alle funzioni di una libreria sottostante mettendone a disposizione un'interfaccia per un diverso linguaggio di programmazione.

di Lua perché questa deve corrispondere a quella dell'interprete Lua interno a LuaTEX attualmente alla 5.2.

Procediamo adesso con il preparare il package manager di Lua chiamato Luarocks, che in passato mi ha dato qualche problema. Consiglio allora di utilizzare l'ultima versione disponibile, a oggi la 2.4.2, scaricando e compilando i sorgenti. I dettagli si trovano alla pagina https://github.com/luarocks/luarocks/wiki/Download.

Scarichiamo e decomprimiamo i sorgenti con i comandi da digitare su un unica riga:

```
$ wget http://luarocks.github.io/luarocks/ \
  releases/luarocks-2.4.2.tar.gz
$ tar -xvzf luarocks-2.4.2.tar.gz
```

Impostiamo la directory di lavoro della finestra di terminale alla cartella decompressa e diamo i classici comandi di costruzione:

```
$ cd luarocks-2.4.2
$ ./configure
$ make build
$ sudo make install
```

Sul sito di ZeroMQ alla pagina http://zeromq.org/bindings:lua vengono citati due progetti che si occupano di fornire agli utenti il binding per Lua. Dopo qualche riflessione ho scelto il progetto `lzmq` ospitato su GitHub alla pagina https://github.com/zeromq/lzmq. In poco tempo, leggendo la documentazione si riesce facilmente a trovare i due ultimi comandi:

```
$ sudo luarocks install lua-llthreads2
$ sudo luarocks install lzmq
```

8.3.3 Prova del funzionamento

Terminate le operazioni d'installazione, conviene eseguire un piccolo script in Lua che stampa la versione di ZeroMQ e i nomi delle funzioni del primo livello della tabella che contiene `lzmq`:

```
-- lzmq test script
local zmq = require "lzmq"

local zver = table.concat(zmq.version(), '.')
print("0MQ version: " .. zver)
print()
print(" -> Main level function:")
for k, v in pairs(zmq) do
    if type(v) == "function" then
        print(k.."()")
    end
end
```

Se si ottiene un output simile al seguente, è molto probabile che ZeroMQ con il suo binding Lua `lzmq` sia stato installato correttamente:

```
0MQ version: 4.1.4

 -> Main level function:
msg_init_data_array()
msg_init_data()
z85_decode()
init_socket()
curve_keypair()
error()
has()
strerror()
msg_init_data_multi()
poller()
assert()
init()
msg_init()
context()
proxy()
msg_init_size()
proxy_steerable()
init_ctx()
z85_encode()
version()
device()
```

8.3.4 Altre vie

Più facile è la procedura d'installazione di ZeroMQ e più lo sarà usare le sue stupefacenti funzionalità da parte degli utenti TEX. Per far si che ZeroMQ sia disponibile vi sono almeno altre due strade.

La prima è utilizzare `swiglib` http://www.luatex.org/swiglib.html un progetto del team LuaTEX creato appositamente per facilitare l'uso di librerie esterne con questo motore di composizione — maggiori informazioni sul progetto si trovano nell'articolo Scarso (2013a). Esso si basa sul programma Swig http://www.swig.org/, un tool di sviluppo che, sulla base dei file di header e di un file di configurazione — non semplicissimo da scrivere come è ovvio attendersi —, crea un sorgente in C pronto da compilare per la creazione del binding a una data libreria C o C++ e per un dato linguaggio.

La seconda strada è quella di avvalersi del modulo FFI presente in LuajitTEX attraverso il progetto `lua-zmq` https://github.com/Neopallium/lua-zmq.

8.4 Shell editors

Ho editato i sorgenti pdfLATEX di questo lavoro con un editor che ormai uso da qualche settimana: Visual Code Editor di Microsoft https://code.visualstudio.com/, installandone la versione per Linux.

In questi editor avanzati multipiattaforma ed estensibili per la scrittura di codice sorgente — Atom ne è un altro esempio — non ricerco particolari utilità rivolte alla compilazione TEX o alla *code completion* del formato LATEX, ma uno spazio di lavoro il più confortevole possibile per l'editazione del testo, ed è abbastanza sorprendente come gli sviluppatori siano riusciti in questi anni a incrementare l'esperienza d'uso degli editor per testi, un tipo di programma che si pensava non avesse più niente di nuovo da dire.

Non esiste l'editor ideale per il sistema TEX ma se ne possono sempre utilizzare due a seconda dell'attività che si sta svolgendo sui sorgenti: costruzione e modifica del testo con un editor avanzato, compilazione e revisione finale delle bozze con uno shell editor per TEX, e TEX Works funziona benissimo per questo compito.

9 FFI e Lua API

9.1 FFI di LuaTEX via Rust

Come esempio dimostrativo della tecnologia FFI mostrerò in questa sezione applicativa come compilare una libreria dinamica in Rust, il linguaggio relativamente nuovo per la programmazione di sistema patrocinato da Mozilla Foundation, per eseguirne l'unica funzione in LuaTEX.

Il sorgente in Rust ha solo quattro righe:

```
#[no_mangle]
pub extern fn double_input(n: i32) -> i32 {
    n * 2
}
```

L'unica funzione, chiamata `double_input()`, è stata marcata con la direttiva `no_mangle` in modo che il compilatore non ne modifichi il nome nel file binario, operazione che in pratica azzera il rischio di collisioni degli identificativi. La keyword `extern` indentifica la funzione come aderente alle specifiche ABI, acronimo di Application Binary Interface.

Caratteristica di Rust è che ogni cosa è un'espressione, persino l'intero corpo di una funzione, perciò non è necessario aggiungere l'istruzione `return` con il doppio dell'argomento, un intero con segno a 32 bit.

Per ottenere il file della libreria è sufficiente fare ricorso a `cargo`, il package manager ufficiale fornito assieme agli altri strumenti di Rust, scrivendo nel file `Cargo.toml` (posizionato nella directory principale del progetto) il codice nel formato TOML:

```
[package]
name = "lib"
version = "0.1.0"
authors = ["your name <yourmail@amail.com>"]

[lib]
name = "double_input"
crate-type = ["dylib"]
```

Di importante in questo *manifest* è il valore `dylib` che sta per *dynamic library*, fornito alla chiave `crate-type`.

Con il comando:

```
$ cargo build
```

il package manager si occuperà di risolvere le eventuali dipendenze — assenti nel nostro esempio —, di compilare il sorgente e di creare il file corrispondente alla libreria dinamica per il dato sistema operativo. Lavorando in Linux e Mac, questo file avrà estensione `.so` che sta per *shared object*, mentre sotto Windows avrà l'estensione `.dll` che sta per *dynamic link library*.

Tornando a LuaTEX, o meglio a LuaLATEX, possiamo adesso caricare questa libreria dinamica via FFI:

```
% !TeX program = LuaLaTeX--shell-escape
\documentclass{minimal}
\begin{document}
\directlua{
local ffi = require('ffi')

ffi.cdef[[
int32_t double_input(int32_t input);
]]

local lib = ffi.load("doubleinput.so")
local res = lib.double_input(12)

tex.print("12 * 2 = " .. res)
}
\end{document}
```

Come si può notare leggendo il listato, usare le funzionalità FFI è abbastanza semplice. Quel che occorre fare è fornire al modulo `ffi` la firma delle funzioni che si intendono usare della libreria con la funzione `cdef()`, e poi caricare la libreria con la funzione `load()` che restituisce il riferimento `lib`, contenente la funzione esterna nella chiave `double_input`.

Quel che invece è più impegnativo è capire quali tipi del C corrispondano a quelli della libreria dinamica. Nel caso dell'esempio si tratta di trovare la corrispondenza con il tipo intero a 32 bit con segno `int32_t`.

Come è logico aspettarsi, è un terreno questo che coinvolge la programmazione di basso livello e che richiede ottime conoscenze tecniche in particolare in quelle situazioni in cui gli oggetti binari producono crash del programma senza alcun messaggio d'errore.

9.2 L'API di Lua via Rust

Lavorando direttamente con l'API di Lua si nota che la maggior parte delle sue funzioni hanno come primo argomento il puntatore all'oggetto `Lua State` che offre l'accesso allo stack d'interfaccia. Inoltre, esse restituiscono sempre un intero che rappresenta il numero di dati inseriti nello stack.

Per l'esempio dimostrativo di questa tecnologia descriverò passo-passo il processo di creazione e compilazione della libreria affinché si possa facilmente ripetere l'esperimento una volta si sia installato Rust sul proprio sistema.

Nel seguente sorgente il codice ha lo scopo di estendere Lua con due funzioni chiamate `magic()` e `sin()`. La prima funzione dimostra la creazione di una tabella Lua attraverso le funzione dell'API e la seconda dimostra un calcolo basato sull'uso dei numeri in virgola mobile. Il listato è suddiviso

in tre parti: l'iniziale definizione dei simboli che corrispondono alle funzioni dell'API di Lua che saranno disponibili a runtime, la definizione delle funzioni utili e la predisposizione della funzione di inizializzazione dal formato fissato dalle specifiche Lua, che verrà eseguita al caricamento della libreria dinamica:

```
#![allow(non_camel_case_types)]
#![allow(non_snake_case)]
#![allow(dead_code)]

extern crate libc;
use libc::{c_int, c_char, c_double};

use std::ffi::CString;

pub type lua_State = libc::c_void;

type lua_Integer = libc::ptrdiff_t;
type lua_Number = c_double;


// Type for native functions that can be
// passed to Lua.
type lua_CFunction = Option<
    unsafe extern "C"
    fn(L: *mut lua_State) -> c_int
>;

#[repr(C)]
struct luaL_Reg {
  name: *const c_char,
  func: lua_CFunction,
}

extern "C" {
    fn lua_createtable(
        L: *mut lua_State,
        narr: c_int,
        nrec: c_int
    );
    fn lua_pushinteger(
        L: *mut lua_State, n: lua_Integer
    );
    fn lua_settable(
        L: *mut lua_State, idx: c_int
    );
    fn lua_pushnumber(
        L: *mut lua_State, n: lua_Number
    );
    fn luaL_setfuncs(
        L: *mut lua_State,
        l: *const luaL_Reg,
        nup: c_int
    );
    fn luaL_checknumber(
        L: *mut lua_State,
        arg: c_int
    ) -> lua_Number;
}

// This test function creates the
// Lua table {5, 10, 15}
unsafe extern "C" fn magic(L: *mut lua_State)
    -> c_int {
    lua_createtable(L, 3, 0);

    lua_pushinteger(L, 1);
    lua_pushinteger(L, 5);// 5
    lua_settable(L, -3);

    lua_pushinteger(L, 2);
    lua_pushinteger(L, 10);// 10
    lua_settable(L, -3);

    lua_pushinteger(L, 3);
    lua_pushinteger(L, 15);// 15
    lua_settable(L, -3);
    1
}

unsafe extern "C" fn sin(L: *mut lua_State)
    -> c_int {
    let x = luaL_checknumber(L, 1);
    lua_pushnumber(L, x.sin());
    1
}

// the format of this function name is
// defined by the Lua manual; the Lua
// interpreter will call into this when you
// require() this library
#[no_mangle]
pub extern "C" fn luaopen_libmagic(
        L: *mut lua_State
    ) -> c_int {
    unsafe { lua_createtable(L, 0, 1); }

    let fn_magic = CString::new("magic")
        .unwrap();
    let fn_sin = CString::new("sin")
        .unwrap();

    let reg = [
        luaL_Reg {
            name : fn_sin.as_ptr(),
            func : Some(sin)
        },
        luaL_Reg {
            name: fn_magic.as_ptr(),
            func: Some(magic)
        },
        luaL_Reg {
            name: std::ptr::null(),
            func: None
        },
    ];
    unsafe {
        luaL_setfuncs(L, reg.as_ptr(), 0);
    }
    1
}
```

Al sorgente in Rust si aggiunge il seguente file di *manifest* `Cargo.toml` perché per mezzo del package manager si possa compilare correttamente la libreria dinamica:

```
[package]
name = "magic"
```

```
version = "0.1.0"
authors = ["yourname <yournick@mail.com>"]

[dependencies]
libc = "0.2.26"

[lib]
crate-type = ["dylib"]
```

Al termine della compilazione possiamo verificare in LuaTEX il funzionamento della libreria di prova scritta secondo l'API nativa di Lua. Per l'utente Lua esse appaiono come normali funzioni eseguibili attraverso i nomi assegnati nella fase di apertura del file binario da parte di `require()`.

```
% !TeX program = LuaTeX
\directlua{
local libmagic = require "libmagic"

local t = libmagic.magic()
tex.print(t)
tex.print([[ ]])
tex.print(libmagic.sin(0.56))
}
\bye
```

10 QRCode Server

In questa sezione e nelle successive due riporterò alcune osservazioni sull'implementazione dei server, ovvero dei programmi connessi tramite protocollo standard a oggetti Socket della libreria ZeroMQ. Tutti i server sono scritti in Rust, un linguaggio molto diverso da Lua, ma che è possibile far interagire con LuaTEX con la tecnologia dello scambio di messaggi neutri, per estenderne le capacità.

Passando alla costruzione del server, il punto di vista sarà quello opposto rispetto al client, che richiede elaborazioni senza conoscere il componente che effettivamente le esegue.

Il QRCode server, di cui è riportata una sessione di lavoro nella figura 8, utilizza il progetto `https://github.com/kennytm/qrcode-rust` per la codifica QRCode e il progetto `rust-zmq` già introdotto alla sezione 4.1 per l'esecuzione delle funzione native di ZeroMQ.

Per questo server, come per gli altri due, il formato dei messaggi multiparte è definito nella corrispondente sezione precedente che descrive gli esempi di utilizzo in LuaTEX.

10.1 Un server robusto

Il server è progettato per essere robusto, ovvero per continuare a funzionare anche nel caso in cui si verifichino errori di comunicazione oppure conseguente al contenuto del messaggio. Come in C, il punto d'ingresso dell'esecuzione del programma è la funzione `main()`. Essa si basa su un ciclo infinito, all'interno del quale per ogni messaggio ricevuto viene elaborata la risposta e inviata al mittente:

```
extern crate qrcode;
extern crate zmq;

use qrcode::QrCode;
use std::error::Error;
use zmq::{Context, Socket, Message, SNDMORE};

fn main() {
    let ctx = Context::new();
    let skt = ctx.socket(zmq::REP)
                    .unwrap();
    assert!(
        skt.bind("tcp://*:5556")
            .is_ok()
    );
    let mut msg = Message::new()
                            .unwrap();
    println!("Server ready on port 5556...");
    loop {
        // get the task object
        let task = match
        recv_and_build_task(&skt, &mut msg) {
            Ok(task) => task,
            Err(e) => {
                match
                send_err(&skt, e.as_str()) {
                    Ok(_) => {},
                    Err(e) =>
                    println!("Error [{}]", e)
                }
                continue
            }
        };

        match send_result(&skt, &task) {
            Ok(_) => {},
            Err(e) => println!("Err [{}]", e)
        }
    }
}
```

La funzione `recv_and_build_task()` si occupa di ricevere il messaggio dal client. Se non ci sono stati errori nella ricezione oppure nel contenuto del messaggio multiparte rispetto al formato atteso, essa restituisce l'oggetto di tipo `QrTask`, una `struct` con la sequenza di zeri e uno del testo già decodificato:

```
fn recv_and_build_task(
    sk: &Socket, msg: &mut Message
    ) -> Result<QrTask, String> {
    // get the message
    let mut multipart = vec![];
    loop {
        match sk.recv(msg, 0) {
            Ok(_) => {},
            Err(e) => return Err(
                e.description().to_string()
            )
        }

        let part =
        if let Some(s) = msg.as_str() {
            s.to_string()
```

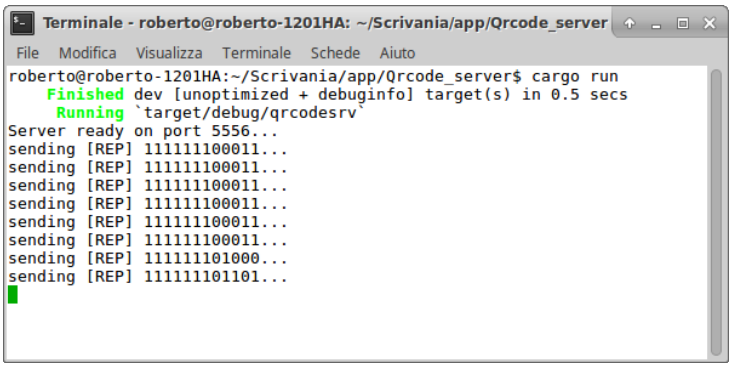


FIGURA 8: Immagine del terminale presa durante l'esecuzione del server che fornisce la codifica QRCode di stringhe di testo. Ogni volta che il server risponde a un messaggio, stampa in console un testo di notifica sia in caso di successo sia in caso di errore.

```
        } else {
            return Err(
                "Malformed UTF-8 command"
                .to_string()
            )
        };
        multipart.push(part);
        if !msg.get_more() {
            break
        }
    }
    if multipart.len() != 3 {
      return Err(
      "Unexpected number of message's frames"
      .to_string()
      )
    }
    if multipart[0] != "QRCODE" {
       return Err(
       "The first frame 'QRCODE' is required"
       .to_string()
       )
    }
    if multipart[2] != "END" {
       return Err(
           "The last frame 'END' is required"
           .to_string()
       )
    }
    QrTask::new(multipart[1].as_str())
}
```

In Rust, come anche in Go — altro moderno linguaggio sviluppato da Google —, non troviamo il paradigma della programmazione a oggetti. Tuttavia è possibile aggiungere molto semplicemente il metodo `new()` a `QrTask` tramite il costrutto `impl`. In questo caso il metodo *costruttore* decodifica il testo nella stringa di zero e uno a rappresentare i quadratini del simbolo QRCode rispettivamente spento o acceso, restituendo l'oggetto oppure un errore tramite il tipo `Result` di cui accenno al termine della sezione 8.2.1:

```
struct QrTask {
    qrenc : String,
}

impl QrTask {
    fn new(code: &str)
    -> Result<QrTask, String> {
        let qrcode = match
        QrCode::new(code.as_bytes()) {
            Ok(qrc) => qrc,
            Err(e) =>
                return Err(format!("{}", e))
        };
        let qrenc = qrcode.render::<char>()
                    .quiet_zone(false)
                    .module_dimensions(1, 1)
                    .light_color('0')
                    .dark_color('1')
                    .build();
        Ok(QrTask { qrenc : qrenc })
    }
}
```

Non rimangono che le funzioni di invio dei messaggi. In caso di esito corretto viene chiamata la funzione `send_result()` mentre in caso si sia verificato qualche errore viene chiamata `send_err()` con una descrizione breve di cosa è andato storto. Sarà responsabilità del client gestire o ignorare l'eventuale messaggio d'errore:

```
fn send_result(sk:&Socket, task: &QrTask)
    -> zmq::Result<()> {
    println!(
        "sending REPLAY {}...",
        &(task.qrenc)[..12]
    );

    // frame 0: <id>
    let e0 = sk.send_str("QRCODE", SNDMORE);
    if e0.is_err() {
        return e0
    }

    let e1 = sk.send_str(
        task.qrenc.as_str(), SNDMORE
    );
    if e1.is_err() {
        return e1
    }
```

```
    sk.send_str("END", 0)
}

fn send_err(sk: &Socket, errmsg: &str)
    -> zmq::Result<()> {
    println!("sending [ERR] {}", errmsg);
    // frame 0
    let e0 = sk.send_str("ERR", SNDMORE);
    if e0.is_err() {
        return e0
    }
    // frame 1
    let e1 = sk.send_str(errmsg, SNDMORE);
    if e1.is_err() {
        return e1
    }
    sk.send_str("END", 0) // frame 2
}
```

11 Prime Server

Il secondo server che descriverò in dettaglio sfrutta la genericità dei messaggi per fornire più di un servizio, in particolare per elaborare dati sui numeri primi. Anche in questo caso il server è implementato in Rust con lo scopo di mantenersi attivo anche in caso di errori.

Volendo evidenziare le parti di codice interessanti di questo server, ci concentreremo solamente su come sono implementate le tre diverse funzioni della verifica di primalità, del conteggio dei primi in un intervallo e la restituzione della lista dei primi in un dato intervallo.

L'idea è che l'implementazione sia tale da consentire l'aggiunta di nuove funzioni al server senza alterare il corpo principale del codice. Ciò è realizzabile solamente se si rende astratto il tipo che rappresenta una funzione e questo in Rust è possibile utilizzando la tecnologia chiamata *object trait* per cui un oggetto è tenuto a mettere a disposizione certe funzioni identificate precisamente da un costrutto `trait`. La gestione di quest'oggetto sarà poi resa indiretta con l'uso di puntatori.

Il compilatore Rust impone vincoli piuttosto stretti al comportamento degli oggetti che si vuole siano oggetti interfaccia, mentre a runtime la tecnica chiamata *dynamic dispatch* basata su una tabella di funzioni, individuerà il metodo e l'oggetto nascosto dietro al tipo interfaccia.

Il *trait* che rappresenta la generica elaborazione sul server chiamato `TaskData` definisce due sole funzioni:

```
trait TaskData {
    fn get_name(&self) -> &str;
    fn resolve(&self, &[bool]) -> Vec<u64>;
}
```

Un oggetto qualsiasi che implementi `TaskData` dovrà definire le funzioni con gli stessi argomenti e con gli stessi tipi di uscita; in altre parole, con la stessa firma.

La funzione `get_name()` è utilizzata per definire il contenuto del primo frame del messaggio di risposta. Si tratta di una funzione perché a oggi Rust non consente che il *trait* possa definire tipi che non siano funzioni. Il metodo `resolve()` è invece quello centrale. Esso riceve come argomento uno *slice* di valori booleani che rappresenta la primalità dei numeri fino a un valore massimo e restituisce un vettore di interi a 64 bit senza segno.

Se per esempio è richiesta la lista dei primi in un intervallo, il vettore risultato conterrà in ordine crescente tale elenco mentre se è richiesto il conteggio dei primi, il vettore conterrà un solo numero corrispondente. In questo modo, non particolarmente elegante in Rust, siamo in grado di eseguire diverse elaborazioni mantenendo per le funzioni la stessa firma.

La verifica della primalità si può aggiungere al server introducendo il nuovo tipo `PCheck` — una `struct` che incorpora il numero che occorre verificare — definendone il costruttore `new()` che, per la regola fissata, si aspetta l'argomento incluso in un vettore di un unico elemento, e infine implementando il `trait TaskData`:

```
struct PCheck<'a> {
    name : &'a str,
    n    : u64
}
impl<'b> PCheck<'b> {
    fn new<'a>(args: Vec<u64>)
    -> Result<Box<PCheck<'a>>, &'a str> {
        if args.len() == 1 {
            Ok(Box::new(PCheck {
                name : "PRIME-CHECK",
                n : args[0]
            }))
        } else {
            Err("Wrong number of arguments")
        }
    }
}
impl<'c> TaskData for PCheck<'c> {
    fn get_name(&self) -> &str {
        self.name
    }
    fn resolve(&self, signal: &[bool])
    -> Vec<u64> {
        let isprime = match self.n {
            0 | 1 => false,
            2     => true,
            x if x % 2 == 0 => false,
            x => {
                signal[(x as usize - 3)/2]
            }
        };
        // [0] == false, [1] == true
        if isprime {
            vec![1]
        } else {
            vec![0]
        }
    }
}
```

Il metodo `resolve()`, in particolare, tratta prima il caso in cui si richieda di verificare 0 e 1, poi il numero 2 e i numeri pari e per ultimo il resto dei numeri dispari. La sequenza calcolata di valori booleani che traccia i numeri primi considera solo i valori dispari a partire da 3, perciò l'intero x è primo oppure no a seconda che nel vettore all'indice $(x-3)/2$ è contenuto `true` o `false`.

A questo punto non rimane che scrivere il metodo costruttore per l'oggetto interfaccia sulla base della chiave contenute nel frame 0 del messaggio:

```
impl TaskData {
    fn newtask(cmd: &str, arg: Vec<u64>)
    -> Result<Box<TaskData>, &str> {
        match cmd {
            "PRIME-CHECK" => {
                match PCheck::new(arg) {
                    Ok(t) =>
                      Ok(t as Box<TaskData>),
                    Err(s) => Err(s)
                }
            },
            "PRIME-LIST" => {
                match PList::new(arg) {
                    Ok(t) =>
                      Ok(t as Box<TaskData>),
                    Err(s) => Err(s)
                }
            },
            "PRIME-COUNT" => {
                match PCount::new(arg) {
                    Ok(t) =>
                      Ok(t as Box<TaskData>),
                    Err(s) => Err(s)
                }
            },
            _ => {
                println!("{}", cmd);
                Err("Unrecognized command")
            }
        }
    }
}
```

Qualsiasi sia l'oggetto che implementa un dato tipo di elaborazione, la seguente funzione `send_result()` sarà sempre la stessa, ricevendo infatti un riferimento *trait object* risolto a runtime poiché a tempo di compilazione si conoscono solamente le funzioni definite dall'interfaccia:

```
fn send_result(
        sk   : &Socket,
        prime: &[bool],
        tk   : Box<TaskData>
    ) -> zmq::Result<()> {
    let res = tk.resolve(prime);
    let n = res.len();
    println!("[OK] sending {} numbers", n);

    // frame 0: <id>
    let e1 = sk.send_str(
        tk.get_name(), SNDMORE
    );
    if e1.is_err() {
        return e1
    }

    for n in res {// result
        let s = n.to_string();
        let e2 = sk.send_str(
            s.as_str(), SNDMORE
        );
        if e2.is_err() {
            return e2
        }
    }
    sk.send_str("END", 0)
}
```

Per completezza riporto la funzione che costruisce il vettore booleano che contiene la sequenza di primalità vero/falso sui numeri dispari maggiori di 3. Essa implementa l'algoritmo del setaccio di Eulero, descritto in fondo alla pagina web https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes:

```
// the primes' signal with the Euler's sieve
fn primes_signal(n: u64) -> Vec<bool> {
    let n = n as usize;
    let lim = (n as f64).sqrt() as usize;
    let mut p_nums = vec![true; (n-1)/2];
    for i in 0..(lim-1)/2 {
        if p_nums[i] {
            let p = 2*i + 3; // prime value
            for k in i..(n/p - 1)/2 {
                p_nums[(p*(2*k + 3) - 3)/2] =
                false;
            }
        }
    }
    p_nums
}
```

12 SQLite Server

La connessione al database SQLite dal programma che svolge il ruolo di server è implementata utilizzando il progetto https://github.com/jgallagher/rusqlite, oggi alla versione 0.12.

Nella figura 9 è riportato il frammento di codice Rust più interessante perché risolve brillantemente il problema di convertire i tipi che restituisce il database in conseguenza all'esecuzione di una query, nel tipo testo con chiavi iniziali fisse a quattro caratteri, per l'invio all'interno del messaggio di risposta al client.

Molto elegantemente è sufficiente implementare il `trait FromSql` previsto in `rusqlite`. Interessante è anche dare uno sguardo al codice che esegue il prepared statement verso il database, racchiuso all'interno di una funzione che in un'unica fase esegue la query e invia il messaggio al client:

```
fn send_query_result(
   sk  : &Socket,
   conn: &Connection,
```

```
struct UniType {
    s: String,
}
impl FromSql for UniType {
    fn column_result(value: ValueRef)
    -> FromSqlResult<Self> {
        let ut = match value {
            ValueRef::Null       => UniType {s : "NULL:".to_string()},
            ValueRef::Integer(i) => UniType {s : format!("INTE:{}", i.to_string())}, // i64
            ValueRef::Real(f)    => UniType {s : format!("REAL:{}", f.to_string())}, // f64
            ValueRef::Text(s)    => UniType {s : format!("TEXT:{}", s.to_string())}, // &str
            ValueRef::Blob(v)    => {
                let s = v.iter().fold(String::new(),
                        |acc, byte| acc + format!("{:03}", byte).as_str()
                );
                UniType {s : format!("BLOB:{}", s)} // &[u8]
            },
        };
        Ok(ut)
    }
}
```

Figura 9: Porzione di codice Rust per l'implementazione del server connesso a un database SQLite. Si tratta del semplice meccanismo con cui si realizza la conversione personalizzata dai cinque tipi previsti dal database al tipo testo adatto alla trasmissione via Socket.

```
   args: &[String]) {
   let query = args[0].as_str();
   // create prepared statement
   let mut stmt = match conn.prepare(query) {
       Ok(stmt) => stmt,
       Err(e) => {
           send_err(sk, e.description());
           return
       }
   };
   let cols = {
       let c = stmt.column_names();
       let mut cols = vec![];
       for cn in c {
           cols.push(cn.to_string());
       }
       cols
   };
   let cols_num = cols.len();
   let mut param: Vec<&ToSql> = vec![];
   for i in 1..args.len() {
       param.push(&args[i] as &ToSql);
   }
   let mut rows =
   match stmt.query(&param[..]) {
       Ok(r) => r,
       Err(e) => {
           send_err(sk, e.description());
           return
       }
   };
   let mut tbl: Vec<String> = vec![];
   while let Some(result_row) = rows.next() {
       match result_row {
           Ok(row) => {
               for i in 0..cols_num as i32 {
                   match row.get_checked(i) {
                       Ok(val) => {
                           let ut: UniType =
                           val;
                           tbl.push(ut.s);
                       },
                       Err(e) => {
                           send_err(&sk,
                             e.description()
                           );
                           return;
                       }
                   }
               }
           },
           Err(e) => {
               send_err(&sk,
                   e.description()
               );
               return;
           }
       }
    }
   // send frames
   send_table(sk, cols, tbl)
}
```

13 Conclusioni

Il programma di composizione LuaTEX è in grado di produrre report molto sofisticati per qualità tipografica e capacità di elaborare i contenuti grazie al linguaggio di alto livello Lua incorporato e ai moduli interni avanzati dedicati ai font e ai nodi.

Oltre a questo, LuaTEX è capace di avvalersi di tecnologie come quelle illustrate in questo lavoro, che mettono in grado il motore di composizione di reperire le informazioni da diverse fonti e in particolare da database relazionali.

Il vantaggio che ne ricava l'utente è duplice: da un lato è possibile ottenere documenti di ottima qualità che si adattano facilmente all'evoluzione delle necessità, e dall'altro si ha a disposizione uno strumento in grado di integrarsi all'interno dell'organizzazione produttiva, che sempre più fa dell'affidabilità e della sicurezza dei dati uno dei motori del proprio sviluppo.

In questo lavoro ho ripercorso alcune possibili tecnologie che estendono LuaTEX con funzioni di acquisizione dell'informazione, esplorando in particolare quella della comunicazione tra codice in esecuzione in differenti processi basata sulla libreria ZeroMQ.

Rendere il codice capace di comunicare indipendentemente da quale sia il linguaggio di programmazione, il sistema operativo o il luogo delle rete dove si trovano i servizi software, apre un entusiasmante e ricco scenario a cui oggi grazie a LuaTEX si aggiungono anche TEX e le sue eccezionali doti tipografiche.

14 Ringraziamenti

Per il suo aiuto, e soprattutto per la sua amicizia, ringrazio Luigi Scarso, senza il quale — lo so, sembra una frase fatta ma vi assicuro che non lo è — questo articolo non sarebbe stato scritto.

Ringrazio infine la redazione tutta di ArsTEXnica per l'impegno profuso nell'accettazione e nella revisione dell'articolo e gli organizzatori del meeting annuale dell'associazione.

Riferimenti bibliografici

Blandy, J. e Orendorff, J. (2016). *Programming Rust: Fast, Safe Systems Development*. O'Reilly Media, 1ª edizione. URL https://www.nostarch.com/Rust.

Dworak, A., Charrue, P., Ehm, F., Sliwinski, W. e Sobczak, M. (2011). «Middleware Trends And Market Leaders 2011». *Conf. Proc.*, **C111010** (CERN-ATS-2011-196), p. FRBHMULT05. 4 p. URL http://cds.cern.ch/record/1391410.

Giacomelli, R. (2016). «LuaTEX + pdfliteral». *ArsTEXnica*, (22), pp. 27–43. URL http://www.guitex.org/home/numero-22.

— (2017). «A database experiment with LuajitLATEX». *ArsTEXnica*, (23), pp. 12–34. URL http://www.guitex.org/home/numero-23.

Giacomelli, R. e Pignalberi, G. (2015). «Generare documenti LATEX con diversi linguaggi di programmazione». *ArsTEXnica*, (20), pp. 40–73. URL http://www.guitex.org/home/numero-20.

Hagen, H. e Scarso, L. (2017). «Foreign function interface in luaTEX». *ArsTEXnica*, (23), pp. 70–76. URL http://www.guitex.org/home/numero-23.

Hintjens, P. (2013a). *Code Connected Volume 1*. iMatix Corporation, 1ª edizione. URL http://hintjens.wdfiles.com/local--files/main:files/cc1pe.pdf.

— (2013b). *ZeroMQ Messaging for Many Applications*. O'Reilly Media, 1ª edizione.

Ierusalimschy, R. (2016). *Programming in Lua*. Lua.org, 3ª edizione.

Klabnik, S. e Nichols, C. (2017). *The Rust Programming Language*. No Starch Press Inc., 1ª edizione. URL https://www.nostarch.com/Rust.

Knuth, D. E. (1984). *The TeXbook*. Addison-Wesley Professional, 1ª edizione.

Scarso, L. (2013a). «Experiments with multitasking and multithreading in LUATEX». *ArsTEXnica*, (16), pp. 68–83. URL http://www.guitex.org/home/numero-16.

— (2013b). «LuajitTEX». *ArsTEXnica*, (15), pp. 59–69. URL http://www.guitex.org/home/numero-15.

The LuaTEX development team (2017). *LuaTEX Reference Manual*, 1.0.4 edizione.

The Rust Community (2017). *The Rust Programming Language*, 2ª edizione.

▷ Roberto Giacomelli
Carrara
giaconet dot mailbox at gmail dot com