PASTEL

Augello Andrea Bafumo Francesco La Martina Marco

23 giugno 2020

Indice

1 Introduzione

PASTEL (Protocol Agnostic Synchronous Transmission Eclectic Language) è stato pensato come un linguaggio di scripting per gestire e coordinare le interazioni tra dispositivi IoT basati su TCP.
L'utilizzo di PASTEL può essere vantaggioso in sistemi ciberfisici complessi che necessitano una conoscenza dello stato globale del sistema per coordinarsi. Infatti il sistema potrebbe includere dispositivi con risorse limitate e scarse capacità computazionali, che quindi non sono in grado di memorizzare lo stato del resto dei sensori ed effettuare decisioni complesse.

Un altro contesto in cui PASTEL può essere utile è testare in modo replicabile il corretto comportamento interattivo di un sistema ciberfisico: il linguaggio proposto infatti può facilmente simulare l'output di molti sensori ed inviarlo agli attuatori.

Il resto della relazione è strutturato come segue: la sezione 2 illustra il contesto in cui si colloca il linguaggio proposto, nella sezione 3 vengono analizzati i dettagli implementativi, nella sezione 4 si illustrano le caratteristiche di PASTEL e degli esempi di applicazioni, infine nella sezione 5 vengono riepilogati i punti principali del lavoro svolto.

2 Stato dell'arte

I dispositivi IoT comunemente eseguono protocolli semplici come CoAP, REST e MQTT [1].

Constrained Application Protocol (CoAP) è un protocollo di trasferimento web specializzato da utilizzare con nodi o reti con risorse limitate (ad es. reti a bassa potenza, senza trasferimento affidabile dei dati). I nodi hanno spesso microcontrollori a 8 bit con piccole quantità di ROM e RAM, mentre reti con risorse limitate come IPv6 su Low-Power Wireless Personal Area (6LoWPANs) hanno spesso elevati tassi di errore dei pacchetti e un tipico throughput di 10 s di kbit/s. Il protocollo è progettato per applicazioni machine-to-machine (M2M) come smart energy e domotica. [2].

Representational State Transfer (REST) è uno tipo di architettura per i sistemi distribuiti. L'espressione representational state transfer e il suo acronimo, REST, furono introdotti nel 2000 nella tesi di dottorato di Roy Fielding, uno dei principali autori delle specifiche dell'Hypertext Transfer Protocol (HTTP), e vennero rapidamente adottati dalla comunità di sviluppatori Internet. Il termine REST rappresenta un sistema di trasmissione di dati su HTTP senza ulteriori livelli, quali ad esempio SOAP. I sistemi REST non prevedono il concetto di sessione, ovvero sono stateless. L'architettura REST si basa su HTTP. Il funzionamento prevede una struttura degli URL ben definita che identifica univocamente una risorsa o un insieme di risorse e l'utilizzo dei verbi HTTP specifici per il recupero di informazioni (GET), per la modifica (POST, PUT, PATCH, DELETE) e per altri scopi (OPTIONS, ecc.) [3].

MQTT è un protocollo di connettività machine-to-machine (M2M) / "Internet of Things". È stato progettato come un trasporto di messaggistica estremamente leggero. È utile per le connessioni remote in cui è richiesto un codice ridotto e/o l'ampiezza di banda della rete è limitata. Ad esempio, è stato utilizzato nei sensori che comunicano con un intermediario tramite collegamento satellitare, attraverso occasionali connessioni dial-up con operatori sanitari e in una gamma di scenari di automazione domestica e piccoli dispositivi. È ideale anche per le applicazioni mobili a causa delle sue dimensioni ridotte, basso consumo energetico, pacchetti di dati ridotti al minimo e distribuzione efficiente delle informazioni a uno o più ricevitori [4].

Il linguaggio proposto può facilmente essere utilizzato con il protocollo REST essendo basato su HTTP, un protocollo di livello applicazione che utilizza solo testo. CoAP, utilizzando UDP come protocollo di livello trasporto, non è compatibile con l'ambiente PASTEL. MQTT si appoggia al TCP, tuttavia è basato sull'invio di codice binario, quindi non è direttamente implementabile in PASTEL a meno di utilizzare particolari tecniche di encoding [5], in quel caso sarebbe certamente possibile, ma non è il tipo di applicazione verso cui è orientato il linguaggio.

L'ambiente target principale è quello dei dispositivi in grado di eseguire codice simbolico [6]. L'utilizzo di codice simbolico (e.g. in dispositivi che eseguono un interprete FORTH [7]) permette di generare in modo automatico istruzioni valide per tutti i dispositivi presi in considerazione, indipendentemente dall'architettura hardware sottostante. Inoltre, essendo possibile definire nuove parole su nodi già installati (utilizzando quelle precedentemente definite) è possibile, a partire da delle primitive legate alle trasmissioni, abilitare tutti i

nodi della rete all'esecuzione di protocolli arbitrari in una fase di inizializzazione in cui viene comunicato il codice simbolico necessario.

Con un approccio simbolico diventa anche possibile supportare protocolli normalmente basati su bytecode come MQTT nativamente [8]. Essendo questo il campo di applicazione principale, su questo si incentreranno di casi d'uso nella sezione 4.5.

PASTEL può essere pensato come un linguaggio per generare programmi che possano mandare e ricevere messaggi testuali tramite TCP, condizionati e iterati facilmente. Questo concept può essere pensato come l'equivalente di `expect` [9, 10, 11] per reti di sensori wireless (WSN).

`expect` è un programma che "parla" con altri programmi interattivi secondo uno script. Seguendo lo script, `expect` sa cosa ci si può aspettare da un programma e quale dovrebbe essere la risposta corretta. Un linguaggio interpretato fornisce ramificazioni e strutture di controllo di alto livello per dirigere il dialogo. Inoltre, l'utente può assumere il controllo e interagire direttamente quando lo desidera, in seguito restituendo il controllo allo script.

Il tool `expect` è basato sul linguaggio Tcl. Tcl fornisce il controllo per il flusso (ad esempio, if, for, break), la valutazione delle espressioni e diverse altre funzioni come la ricorsione, definizione delle procedure, ecc..

Tcl è stato uno dei primi linguaggi "dinamici" a diventare popolare, vedendo un uso diffuso a partire dai primi anni '90. Tcl ha trovato ampio successo in molti campi, e le distribuzioni di Tcl spaziano da dispositivi integrati come router, a infrastrutture di back-end distribuite. [12].

Alla luce di quanto detto, con questo linguaggio ci prefiggiamo l'obiettivo di fornire uno strumento che permette di interfacciarsi con facilità e in modo affidabile con dispositivi IoT, con la possibilità di potere iterare le operazioni, manipolare i dati con estrema facilità e definire delle routine da potere effettuare in maniera condizionata.

3 Descrizione del progetto

Il linguaggio PASTEL nasce con lo scopo di agevolare la programmazione di software che si occupano di gestire dispositivi in ambiente IoT. Viene proposto un interprete per il linguaggio PASTEL, piuttosto che un compilatore, sia per la minore complessità implementativa, che per favorire l'interattività delle operazioni che vengono normalmente svolte in questo contesto. Per dare la possibilità al programmatore di effettuare tutte le manipolazioni ai dati del caso, vi sono quattro tipi primitivi, ovvero `int, real, string` e `address`. La possibilità di poter definire nuovi tipi non è consentita, per non dover appesantire inutilmente il linguaggio, poiché nel contesto in cui viene utilizzato, non è un requisito fondamentale.

Il tipo `address` è il tipo che caratterizza il linguaggio poiché è grazie al suo utilizzo che sono semplificate molte operazioni che un linguaggio da utilizzare in ambiente IoT deve supportare.

Considerando inoltre che PASTEL si pone in un contesto in cui i dati scambiati tra i device sono delle sequenze di caratteri, non poteva che essere presente il tipo `string`. Le stringhe vengono trattate come un tipo primitivo immutabile e non è consentito l'accesso diretto al singolo carattere. Per potere ottenere i singoli caratteri, può però essere utilizzata la funzione `s2l` che restituisce un valore di tipo `list`, ovvero una lista dei singoli caratteri (rappresentati come stringhe di lunghezza 1) della stringa data in input alla funzione.

La separazione tra `int` e `real` è stata pensata per evitare problemi dovuti all'imprecisione della rappresentazione in virgola mobile [13].

Il tipo `device` è un tipo composto da `address` e `int`, poiché un device è caratterizzato dal suo indirizzo IP e da un numero di porta, dato che il linguaggio poggia le sue basi sul protocollo TCP/IP.

Il tipo `list` viene pensato per rendere facili ed intuitive le operazioni di iterazione. Questo tipo di dato infatti caratterizza liste eterogenee che possono contenere elementi di qualsiasi tipo, tranne un'altra lista. Non è consentito infatti l'uso di liste innestate. È però possibile scrivere una funzione in PASTEL, come quella riportata di seguito, per fare un inserimento con flattening, potendo così inserire gli elementi di una lista all'interno di un'altra.

```
  def flat_insert(a,b,pos){
    int i = 0;
    for(item in b){
        insert(a,item,pos+i);
        i=i+1;
    }
    a;
}
```

Non è presente la definizione degli array, la cui introduzione nel linguaggio sarebbe risultata ridondante poiché tutte le operazioni consentite sugli array possono essere fatte con le liste. È infatti consentita anche la possibilità di accedere agli elementi della lista tramite la loro posizione con i selettori normalmente utilizzati sugli array: "`nome_lista[index]`".
Vi è un grande vantaggio nell'utilizzo delle liste, ovvero la possibilità di iterare per ogni elemento della lista tramite il costrutto `foreach`. La variabile di controllo viene inizializzata all'avvio del ciclo, al termine viene eliminata e non è più referenziabile. Inoltre all'interno del ciclo può essere modificata (e.g. tramite assegnamento).

Per quanto riguarda le funzioni, è possibile sia utilizzare delle funzioni già presenti nel linguaggio, trattate nelle sezione refsubsection:fun, ma è anche possibile definirne di nuove. I nomi delle funzioni built-in sono parole riservate e quindi non è possibile ridefinirle. Anche quelle definite dall'utente per semplicità implementativa non sono ridefinibili e non è consentito definire una funzione all'interno della definizione di un'altra funzione. Non è richiesta la definizione del tipo dei parametri formali, l'onere di gestire la coerenza dei tipi è lasciato al programmatore, che ha quindi la possibilità di creare funzioni che non sono vincolate dai tipi dei parametri formali. Un esempio è mostrato nel listato 1. Inoltre i parametri alle funzioni sono passati per copia e la valutazione di eventuali espressioni passate come parametri reali viene effettuata al momento della chiamata alla funzione, non quando i parametri formali vengono utilizzati nel corpo della funzione(eager evaluation).

Il tipo di una variabile deve essere definito al momento della dichiarazione, e non è consentita la modifica. Questo consente di scrivere programmi più robusti e dal debug più semplice. Inoltre al momento della dichiarazione, alla variabile viene assegnato un valore di default per evitare inconsistenze sui dati.

Per non limitare la potenza del linguaggio, le funzioni definite dall'utente possono restituire tipi di dati diversi (una forma primitiva di overloading indipendente dal contesto). In questi casi si possono utilizzare le liste eterogenee per memorizzare i valori restituiti ed eseguire operazioni diverse in base al tipo, come precedentemente dimostrato nel listato 1.

I valori di ritorno delle funzioni corrispondono all'ultimo valore calcolato all'interno della funzione, senza bisogno di esplicitare direttamente cosa la funzione debba restituire. Questo evita salti e interruzioni dell'esecuzione che peggiorano la leggibilità del codice [14].

Esiste la possibilità di passare file di input (con estensione `.pa`) all'interprete in modo da eseguire i sorgenti. Dopo di ciò viene restituito il controllo all'interprete. Un altro modo per includere file sorgente è quella di utilizzare la direttiva di inclusione, eseguita in fase di lexing per non rendere possibili include condizionali, potenzialmente ambigui e con comportamento imprevedibile.

Le variabili sono definite con scope limitato alla funzione in cui vengono dichiarate. Una funzione può accedere a tutti i simboli definiti in scope più ampi, ma per discriminare i riferimenti a variabili locali dai riferimenti a variabili di scope più ampio, non viene consentita la dichiarazione implicita delle variabili, quindi la ricerca nelle tabelle dei simboli avviene nel momento in cui le espressioni vengono valutate (scoping dinamico), e non durante il lexing, differentemente dall'implementazione suggerita dal libro di testo [15]. Se una variabile viene definita fuori da una funzione avrà scope globale e la sua vita terminerà alla chiusura dell'interprete.

I cicli iterativi condividono lo scope della funzione in cui vengono eseguiti per ragioni di performance, definire quindi una variabile all'interno di un ciclo porta ad una condizione di errore, poiché dalla seconda iterazione la variabile risulterà già definita.

```
1  def conditional_return(unknown_val){
2          string type = typeof(unknown_val);
3          if(type=="int"){
4                  int result = unknown_val;
5          } else { if(type == "device"){
6                  string result = toString(unknown_val);
7          } else {
8                  quit("Unexpected type");
9          }}
10         result;
11 }
12
13 list mixedTypes = [ conditional_return(3), conditional_return(localhost:1) ];
14
15 if(typeof(mixedTypes[0]) == "string"){
16         device first_val = s2d(mixedTypes[0]);
17         print("Found an address!");
18 } else {
19         int first_val = mixedTypes[0];
20         print("Found an int!");
21 }
22
23 if(typeof(mixedTypes[1]) == "string"){
24         device second_val = s2d(mixedTypes[1]);
25         print("Found an address!");
26 } else {
27         int second_val = mixedTypes[1];
28         print("Found an int!");
29 }
30
31 print(first_val);
32 print(second_val);
```

Listing 1: Esempio di codice PASTEL in cui vengono utilizzate funzioni con tipo di ritorno variabile e dichiarazioni condizionali

4 Caratteristiche del linguaggio

4.1 Grammatica

Tipi

Il linguaggio PASTEL possiede sia tipi primitivi che composti. I tipi primitivi sono:

- `int`, utilizzato per rappresentare i numeri interi
- `real`, utilizzato per rappresentare i numeri in virgola mobile
- `string`, utilizzato per rappresentare le stringhe di caratteri
- `address`, utilizzato per rappresentare gli indirizzi IPv4

I tipi composti sono invece:

- `device`, prodotto cartesiano tra i tipi address e int
- `list`, tipo ricorsivo

V1\V2	int	real	string	address	device	list
int	+ - * / or and CMP		*	+ -		
real		+ - / * CMP				
string	*		+ CMP			
address	+ -			CMP		
device					CMP	
list						+

Tabella 1: Operazioni consentite dal linguaggio, sono da intendersi come V1 <operatore> V2. Con CMP si indicano le operazioni di comparazione

Strutture di controllo
Il linguaggio fornisce la possibilità di usare sia comandi condizionali che comandi iterativi. I comandi condizionali sono i classici:

- `if ( condition ) { instructions }`
- `if ( condition ) { instructions } else { instructions }`

I comandi iterativi invece sono:

- `while ( condition ) { instructions }`
- `for ( var in list ) { instructions }`

Nell'espressione `while` si itera indefinitivamente finché la condition è diversa da 0, nel `for` invece si itera per ogni elemento contenuto nella lista.

Operazioni
Il linguaggio consente le seguenti operazioni: somma, sottrazione, moltiplicazione, divisione, and e or logici, e comparazioni (==, !=, >, >=, <, <=). Le combinazioni di tipi consentite per ogni operatore sono riassunte nella tabella 1. Le comparazioni e gli operatori logici restituiscono sempre degli int, tutti gli altri operatori restituiscono un valore del tipo della colonna, tranne la moltiplicazione `int * string` equivalente a `string * int`.

Selettori
Il tipo device, come già accennato precedentemente, è un prodotto cartesiano tra `address` e `int`. Infatti un device è caratterizzato da un indirizzo IP e un numero di porta. Per potere accedere al singolo campo del record, vengono utilizzati i selettori `.address` e `.port`. I campi di un device sono accessibili con i suddetti selettori, ma non è consentito un aggiornamento selettivo, impedendo di modificare un device verso il quale è aperta una connessione. È comunque consentito l'aggiornamento totale. Per la creazione di un valore di tipo device si utilizza la seguente sintassi "`<address>:<port>`".

Funzioni
Il linguaggio PASTEL offre al programmatore un'ampia varietà di funzioni built-in, che verranno illustrate nella sezione 4.4. Chiaramente è anche possibile estendere il linguaggio definendone di nuove tramite la sintassi "`def nome_funzione ( parametri_formali ) { istruzioni }`".

4.2 Parser

Un requisito del progetto è di creare il linguaggio con i tools Bison e Flex [15], due strumenti per costruire programmi che gestiscano input strutturati. Flex si occupa dell'analisi lessicale (lexing) mentre Bison si occupa dell'analisi della sintassi (parsing). L'output dei due tools è un parser di tipo LALR(1) che effettua un parsing bottom-up, con cui è possibile gestire produzioni left-recursive, utilizzando Bison per generare il parser e Flex per riconoscere i token nella fase di lexing.

Riportiamo di seguito le regole di produzione in formato BNF del linguaggio proposto:

```
<program>::=
        | <program> <stmt>
        | <program> DEF NAME '(' <symlist> ')' '{' <list> '}'

<stmt>::= IF '(' <exp> ')' '{' <list> '}'
        | IF '(' <exp> ')' '{' <list> '}' ELSE '{' <list> '}'
        | WHILE '(' <exp> ')' '{' <list> '}'
        | FOR '(' NAME IN <exp> ')' '{' <list> '}'
        | <exp> ';'
        | <decl> ';'

<list>::=
        | <stmt> <list>

<exp>::= '(' <exp> ')'
        | FUNC '(' <explist> ')'
        | NAME '(' <explist> ')'
        | VALUE
        | '[' <explist> ']'
        | <exp> ':' <exp>
        | <exp> '+' <exp>
        | <exp> '-' <exp>
        | <exp> '*' <exp>
        | <exp> '/' <exp>
        | <exp> AND <exp>
        | <exp> OR <exp>
        | <exp> CMP <exp>
        | '-' <exp>
        | NAME'[' <exp> ']'
        | NAME
        | <exp> ADDR
        | <exp> PORT


<decl>::= TYPE NAME
        | NAME '=' <exp>
        | NAME '[' <exp> ']' '=' <exp>
        | TYPE NAME '=' <exp>

<explist>::=
        | <exp>
        | <exp> ',' <explist>

<symlist>::=
        | NAME
        | NAME ',' <symlist>
```

Le regole di produzione per `<exp>` sembrerebbero ambigue, infatti una stessa sequenza di operatori potrebbe essere rappresentata da più alberi di parsing. È generalmente possibile riscrivere la grammatica per eliminare l'ambiguità, però Bison mette a disposizione degli strumenti che permettono di specificare la precedenza degli operatori disambiguando la grammatica e scartando automaticamente gli alberi di parsing non validi, consentendo l'utilizzo di una grammatica più semplice e compatta.

In figura 1 viene riportato l'automa a stati finiti generato da Bison su una versione ridotta della grammatica per fornire un'idea generale della struttura di un programma grammaticalmente corretto. Questa grammatica semplificata rimuove delle regole di produzione ricorsive per ridurre il numero di stati, miglio-

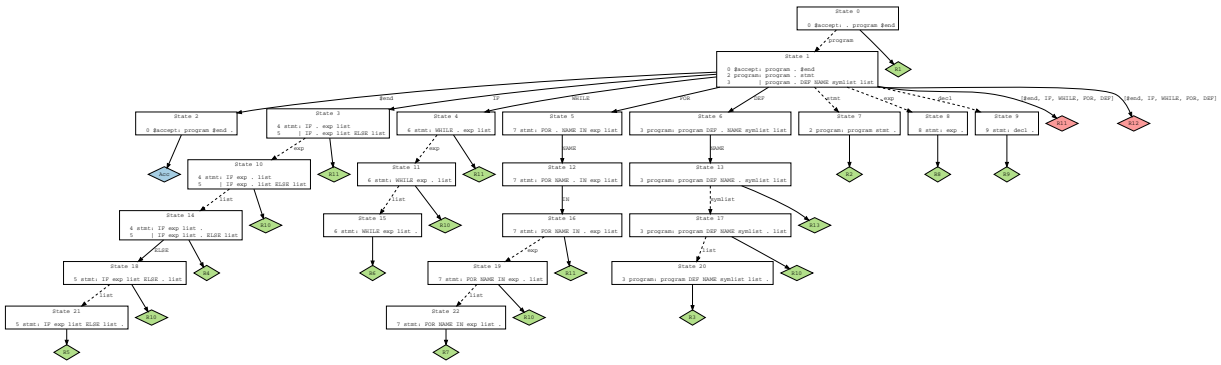

Figura 1: Grafo del FSA generato utilizzando GNU Bison 3.5.4 XML Automaton Report su una struttura fortemente semplificata per facilitare la rappresentazione. La grammatica completa non presenta conflitti shift/reduce e reduce/reduce.

rando la leggibilità. In appendice si riporta il grafo della grammatica completa.

4.3 Lexer

Per quanto riguarda la struttura del lexer, vengono utilizzate delle regex per matchare gli identificatori di ogni tipo del linguaggio, i valori consentiti per ogni tipo, i nomi delle variabili, costanti proprie del linguaggio, l'inserimento di commenti, simboli per la comparazione, operazioni consentite sui tipi, i simboli consentiti, le funzioni built-in del linguaggio, le direttive di preprocessing, i selettori e le strutture di controllo. Queste espressioni regolari sono state utilizzate anche per creare un package di syntax highlighting per l'editor di testo "Atom" disponibile open-source online: https://atom.io/packages/language-PASTEL

Tipi

```
int      "int"
real     "real"
address  "address"
string   "string"
list     "list"
device   "device"
```

Valori

```
int      [0-9]+
real     [0-9]+\.[0-9]+
address  ((([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.)
    ↪ {3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5]))|"localhost"
string   \".*\"
```

Nomi variabili

```
nomi     [a-zA-Z_][a-zA-Z0-9_]*
```

Costanti

```
newline  "NL"
true     "TRUE"
false    "FALSE"
```

Commenti

```
inline   "\\".*
block    "/*"[^*\/]*"*/"
```

Comparazioni

```
maggiore                "<"
minore                  ">"
diverso                 "!="
uguale                  "=="
maggiore o uguale       "<="
minore o uguale         ">="
```

Operazioni

```
and             "and"
or              "or"
somma           "+"
differenza      "-"
moltiplicazione "*"
divisione       "/"
assegnamento    "="
```

Simboli

```
,                       ","
;                       ";"
{                       "{"
}                       "}"
[                       "["
]                       "]"
(                       "("
)                       ")"
spazi e tabulazioni     [ \t]
ritorno a capo          "\n"
altri simboli           .
```

Funzioni built-in

```
print           "print"
quit            "quit"
connect         "connect"
disconnect      "disconnect"
receive         "receive"
send            "send"
insert          "insert"
remove          "remove"
length          "length"
s2i             "s2i"
s2r             "s2r"
s2d             "s2d"
s2a             "s2a"
s2l             "s2l"
toString        "toString"
console         "console"
strip           "strip"
split           "split"
sleep           "sleep"
```

```
typeof          "typeof"
isConnected     "isConnected"
include         ^"#"[ \t]*include[ \t]*[\"].*[\"]
debug           "debug"[0-9]+
```

Selettori

```
.port           ".port"
.address        ".address"
```

Strutture di controllo

```
if      "if"
else    "else
while   "while"
for     "for"
in      "in"
def     "def"
```

Per riuscire a scrivere qualsiasi simbolo all'interno delle stringhe, vengono utilizzati gli `state` di Flex. Le costanti `TRUE` e `FALSE` corrispondono semplicemente a 1 e 0, mentre `NL` rappresenta il carattere di ritorno a capo

4.4 Funzionalità del linguaggio

In PASTEL è presente un'ampia gamma di funzioni che possono essere utilizzate per gestire le connessioni dei devices, manipolare i dati ed eseguire funzioni di sistema. Vengono di seguito elencate e descritte tutte le funzioni. Per indicare il numero di argomenti richiesti da ogni funzione, viene adoperata una notazione Prolog-like.

- `console/1` effettua una chiamata di sistema per aprire una finestra di terminale su cui viene eseguito il programma `netcat` in modalità server, in ascolto sulla porta del device specificato come argomento. Lo scopo di questa funzione è facilitare le operazioni di testing, permettendo di simulare il comportamento di un dispositivo fisico; un altro possibile impiego è consentire all'utente l'inserimento di dati in maniera interattiva. Questa funzione è parametrizzata in base al sistema operativo (Linux o macOS) poiché i due sistemi operativi usano diverse convenzioni. In altri sistemi operativi questa funzione non è supportata.

- `connect/1` attiva una connessione TCP verso il `device` passato come parametro. Se il tentativo di connessione ha successo, una chiamata a `isConnected/1` sullo stesso device restituirà un valore diverso da 0 (TRUE), in caso contrario verrà stampato un messaggio di errore sullo standard error, e chiamate a `isConnected/1` restituiranno FALSE.

- `disconnect/1` termina la connessione TCP precedentemente creata verso il `device` passato come parametro. Se non c'è una connessione attiva verso il device, verrà stampato un messaggio di errore sullo standard error.

- `receive/1` tenta di ricevere una stringa dal device passato come parametro e restituendola come valore di ritorno. Se non è presente una connessione attiva verso il device, verrà stampato un messaggio di errore sullo standard error.

- `send/2` manda al `device` passato come primo parametro della funzione e verso cui è già attiva una connessione TCP, la stringa passata come secondo parametro

- `isConnected/1` restituisce un valore non nullo se c'è una connessione attiva verso il `device` passato come parametro, altrimenti restituisce `0`

- `length/1` restituisce un valore di tipo `int` che rappresenta il numero degli elementi contenuti nella lista passata alla funzione come parametro.

- `insert/3` inserisce nella lista passata come primo parametro l'elemento contenuto nel secondo parametro, nella posizione specificata dal terzo parametro. Quindi se si vuole fare un inserimento in testa, basta usare `insert(nome_lista, elemento, 0)`, per aggiungere in coda invece si può sfruttare la funzione `length` e quindi usare `insert(nome_lista, elemento, length(nome_lista))`. Se, data una lista `l`, `length(l)=N1` e viene eseguita `insert(l,x,N2)` con `N2>N1`, verrà stampato un messaggio di errore sullo standard error, non essendo consentiti elementi di una lista non inizializzati.

- `remove/2` rimuove dalla lista passata come primo parametro, l'elemento nella posizione specificata dal secondo parametro. Se si vuole eliminare il primo elemento della lista si può usare `remove(nome_lista, 0)`, se si vuole eliminare l'ultimo si può ricorrere alla funzione `length` e quindi usare `remove(nome_lista, length(nome_lista)-1)`. Tentare di rimuovere un elemento in una posizione negativa, o oltre la fine della lista, comporta un messaggio di errore.
  Si noti che le funzioni `insert/3` e `remove/2` modificano direttamente la lista passata come primo argomento, non ne restituiscono una copia con le modifiche specificate, queste funzioni hanno quindi effetti collaterali ed è necessario prestare attenzione quando si adoperano.

- `typeof/1` restituisce una stringa rappresentante il tipo del valore passato come parametro.

- `s2i/1` restituisce un valore di tipo `int` a partire da un valore di tipo `string` passato come parametro. Stampa un messaggio di errore nel caso in cui la stringa non rappresenti un valore intero in base 10.

- `s2r/1` restituisce un valore di tipo `real` a partire da un valore di tipo `string` passato come parametro. Stampa un messaggio di errore nel caso in cui la stringa non rappresenti un numero reale.

- `s2a/1` restituisce un valore di tipo `address` a partire da un valore di tipo `string` passato come parametro. Stampa un messaggio di errore nel caso in cui la stringa non rappresenti un indirizzo IP valido.

- `s2d/1` restituisce un valore di tipo `device` a partire da un valore di tipo `string` passato come parametro. Se la stringa non rappresenta una combinazione valida di indirizzo IP e numero di porta separati dal simbolo `':'` viene stampato un messaggio di errore.

- `s2l/1` restituisce un valore di tipo `list` a partire da un valore di tipo `string` passato come parametro. La lista restiuita conterrà i singoli caratteri che formano la stringa di input. Restituisce un errore nel caso in cui l'argomento di input non sia una stringa.
  L'operazione inversa non è fornita come una funzione standard perché, come mostrato nella seguente implementazione *naïve*, è nativamente implementabile in PASTEL:

  ```
  def join_chars(split_string){
    string joined_string = "";
    for(char in split_string){ joined_string = joined_string + char; }
    joined_string;
  }
  ```

- `toString/1` restituisce un valore di tipo `string` a partire da un parametro di un qualsiasi tipo. Nella conversione di una lista in stringa, viene restituita una stringa formata dall'elenco degli elementi che fanno parte della lista separati da virgole e racchiusi da parentesi quadre, ovvero "`[ el_1 , el_2 , ... , el_n ]`".

- `strip/1` restituisce una stringa ottenuta da quella passata come parametro rimuovendo i caratteri di carriage return, line feed, tabulazioni, e spazi a inizio e fine stringa.

- `split/2` restituisce un valore di tipo `list`, contenente tutte le sottostringhe della stringa passata come primo parametro ottenute considerando come divisore dei campi la stringa passata come secondo parametro.

- `print/1` stampa a schermo il risultato della funzione `toString` del parametro, seguito da un ritorno a capo.

- `sleep/1` effettua una chiamata di sistema per sospendere il processo per il numero di secondi passato come parametro.
- `quit/1` effettua una chiamata di sistema per terminare l'esecuzione dell'interprete. Il parametro, se di tipo `int`, serve per specificare un valore di ritorno del programma o, se di tipo `string`, per stampare una stringa a fine esecuzione.

Il linguaggio è inoltre fornito di una modalità `debug`, attivabile scrivendo `debug` seguito da un numero diverso da 0 e disattivabile con `debug0`. Questa modalità consente di stampare a schermo una rappresentazione testuale dell'albero sintattico generato in fase di parsing dopo l'esecuzione delle operazioni.
Un'altra importante funzionalità è la direttiva di inclusione, che fornisce uno strumento fondamentale per la programmazione in grande.

4.5 Casi d'uso

Per verificare l'applicabilità del linguaggio nel contesto target sono state effettuate delle simulazioni di potenziali ambiti in cui PASTEL potrebbe essere adoperato. Poiché il linguaggio presenta numerose funzionalità e permette lo sviluppo di software anche molto complesso, non essendo possibile esplorare tutte le possibilità offerte da PASTEL in modo conciso, ai fini di questa trattazione è stato scelto di mostrare un sottoinsieme delle funzionalità ritenuto particolarmente rappresentativo.

4.5.1 Network discovery

In una rete di sensori generalmente è necessario che ogni nodo sappia quali sono i suoi vicini e come raggiungerli, a questo scopo sono stati sviluppati un grande numero di protocolli [16, 17, 18].

In questo caso d'uso mostriamo come, con un codice relativamente compatto, a partire dalla conoscenza dell'indirizzo di un singolo nodo indicato con `root`, sia possibile ricostruire la topologia della porzione di rete connessa al nodo `root`.
Nel codice mostrato in 2 ogni nodo viene interrogato per ottenere la lista dei suoi vicini tramite la procedura `find_neighbours`. Tutti i vicini del nodo che non sono già stati individuati precedentemente vengono aggiunti ad una lista di nodi noti.

Ogni nodo comunica la lista dei suoi vicini restituiendo in formato testuale indirizzo IP e numero di porta ai quali sono raggiungibili, separando i vari nodi con delle virgole. Il linguaggio PASTEL mette a disposizione funzioni bultin che permettono in modo estremamente semplice, a partire dai dati forniti in questo formato, di ottenere una lista PASTEL e convertire tutte le stringhe in device utilizzabili all'interno del programma.

L'algoritmo termina automaticamente nonappena l'ultimo dei nodi noti interrogato non restituisce nodi nuovi, a quel punto si saranno individuati tutti i nodi raggiungibili a partire da `root`.

Per semplicità si assume che il calcolatore che esegue il programma abbia un range di comunicazione molto superiore ai nodi della rete, e che possa quindi raggiungerli in un singolo hop, una strategia più generale dovrebbe ovviamente prevedere la costruzione di messaggi di routing.

```
1  #include "../testfiles/list_functions.pa"
2
3  device root=127.0.0.1:1234;
4  list known_nodes = [ root ];
5
6
7  def find_neighbours(node){
8                  string temp;
9                  string message="Neighbours_list reply"+NL;
10                 connect(node);
11
12                 if(isConnected(node)==FALSE){
13                         quit("Cannot connect to root at "+root);
14                 }
15                 send(node,message);
```

```
16                  string reply=receive(node);
17                  list neighbours = split(reply,","); //expected CSV
18                  int i=0;
19                  while(i<length(neighbours)){
20                          temp = strip(neighbours[i]);
21                          if(temp!=""){
22                                  neighbours[i]=s2d(temp);
23                                  i=i+1;
24                          } else {
25                                  remove(neighbours,i);
26                          }
27                  }
28                  disconnect(node);
29
30                  for(d in neighbours){
31                          if(inList(d,known_nodes) == FALSE){
32                                  append(known_nodes,d);
33                          }
34                  }
35                  print("Neighbours of device " + toString(node));
36                  print(neighbours);
37  }
38
39  int i = 0;
40  while(i < length(known_nodes)){
41          console(known_nodes[i]);
42          find_neighbours(known_nodes[i]);
43          i = i+1;
44  }
45
46  print("Total found nodes:");
47  print(known_nodes);
48  quit(0);
```

Listing 2: Sorgente network_discovery.pa

4.5.2 Publish-Subscribe

Come visto precedentemente in [1] il modello Publish-Subscribe è molto comune come protocollo di comunicazione nel campo IoT. Nel listato 3 mostriamo come PASTEL permetta con facilità di interrogare una lista di nodi per ottenere i topic a cui ogni nodo intende fare il subscribe, dopo questo step di configurazione è immediato poter mandare a tutti i nodi iscritti ad un topic un determinato messaggio.

```
1   def topicPost(topic, msg){ // sends the msg string to every device in topic
2           for(d in topic){
3                   connect(d);
4                   if(isConnected(d)){
5                           send(d, msg);
6                   disconnect(d);
7                   } else {
8                           print("Couldn't connect to " + toString(d));
9                   }
10          }
11  }
12
```

```
13 list known_devices = [ localhost:1234, localhost:1235, localhost:1236,
       ↪ localhost:1237];
14
15 list topic1;
16 list topic2;
17 list topic3;
18
19 string reply;
20 list topics;
21
22 for(d in known_devices){
23         console(d);                          // sets up simulated environment.
24 }
25
26 for(d in known_devices){
27         connect(d);
28         if(isConnected(d)){
29                 send(d, "SUBS_LIST REPLY TELL: ~ :TELL"+NL);
30                 reply = strip(receive(d));
31                 if(reply!=""){
32                         topics=split(reply,",");
33                         for(t in topics){
34                                 if(strip(t)=="t1"){
35                                         insert(topic1,d,0);
36                                 } else{ if(strip(t)=="t2"){
37                                         insert(topic2,d,0);
38                                 } else{ if(strip(t)=="t3"){
39                                         insert(topic3,d,0);
40                                 }}}
41                         }
42                 }
43         disconnect(d);
44         } else {
45                 print("Couldn't connect to " + toString(d));
46         }
47 }
48
49 sleep(2);
50 print("Topic 1: ");print(topic1);
51 print("Topic 2: ");print(topic2);
52 print("Topic 3: ");print(topic3);
53
54
55 print("Sending ACK to TOPIC1");
56 topicPost(topic1, "TOPIC1 SUBSCRIPTION ACK"+NL);
57 print("Sending ACK to TOPIC2");
58 topicPost(topic2, "TOPIC2 SUBSCRIPTION ACK"+NL);
59 print("Sending ACK to TOPIC3");
60 topicPost(topic3, "TOPIC3 SUBSCRIPTION ACK"+NL);
```

Listing 3: Sorgente subscriber.pa

Con questo framework diventa estremamente facile per un nodo comunicare un messaggio da mandare a tutti i nodi iscritti ad un topic, come mostrato nel listato 4.

Il programma si aspetta un input del tipo "`<lista di topic> :TOPICS <messaggio>`", la parte di messaggio che precede "`:TOPICS`" è interpretata come una lista di topic separati da virgole, il messaggio che segue quel tag, invece, verrà inviato a tutti i nodi che si sono precedentemente iscritti ad almeno uno di quei topic.

```
1  # include "../testfiles/subscriber.pa"
2
3  list post;
4  string message;
5
6  for (d in known_devices){
7          connect(d);
8          if(isConnected(d)){
9                  send(d, "PUBLISH REPLY TELL: ~ :TELL"+NL);
10                 reply= receive(d);
11                 disconnect(d);
12                 if(reply !=""){
13                         post = split(reply, ":TOPICS");
14                         topics = split(post[0],",");
15                         message = strip(post[1]);
16                         for ( t in topics ){
17                                 if  (strip(t)=="t1") {
18                                         topicPost(topic1, message+NL);
19                                 }else{if(strip(t)=="t2") {
20                                         topicPost(topic2, message+NL) ;
21                                 }else{if(strip(t)=="t3") {
22                                         topicPost(topic3, message+NL) ;
23                                 }}}
24                         }
25                 }
26         } else {
27                 print("Couldn't connect to " + toString(d));
28         }
29 }
```

Listing 4: Sorgente publisher.pa

Lo schema di funzionamento illustrato può ovviamente essere ulteriormente raffinato impedendo l'iscrizione multipla ad uno stesso topic e aggiungendo controlli per evitare di inviare più volte uno stesso messaggio un nodo iscritto a più topic, o al nodo che ha originato il messaggio. Inoltre è possibile prevedere la possibilità di cancellare le iscrizioni o modificarle periodicamente.

4.6 Risultati ottenuti

Il codice mostrato nelle sezioni 4.5.1 e 4.5.2 mostra la potenzialità di essere integrato in un unico sistema software che, inserito in una rete, a partire dalla conoscenza di un singolo nodo, riesce a ricostruire la topologia della rete raggiungibile e gestire il funzionamento di un protocollo non banale del tipo publish-subscribe, dimostrando l'applicabilità del linguaggio proposto all'interno del contesto target.

Per testare la scalabilità dell'implementazione del linguaggio abbiamo misurato il tempo impiegato per inviare un numero crescente di messaggi.
Un invio di messaggio si riassume nelle seguenti operazioni:

1. Si stabilisce una connessione.
2. Si verifica che la connessione abbia avuto successo.
3. Si invia il messaggio.

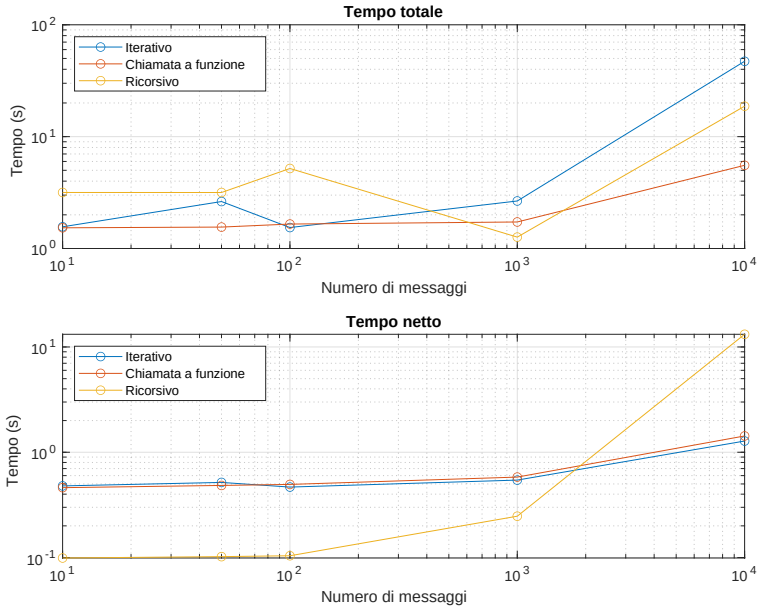

Figura 2: Performance dell'interprete

4. Si richiude la connessione.

Al fine di ottenere risultati il più possibile realistici abbiamo usato tre schemi diversi con cui mandare i messaggi:

- Ciclo iterativo: all'interno di un ciclo while vengono eseguite le operazioni sopra citate.
- Chiamate a funzione: all'interno di un ciclo while viene chiamata una funzione che effettua le operazioni di invio.
- Funzione ricorsiva: una funzione ha un parametro che conta quanti messaggi devono ancora essere inviati. Al termine dell'invio di un messaggio il parametro viene decrementato e si effettua una chiamata ricorsiva.

Il tempo impiegato è misurato utilizzando il programma di sistema `time` [19]. Il tempo totale corrisponde al valore "real", mentre il tempo netto è la somma di "user" e "sys". I programmi eseguono una chiamata a `console()` che comporta un secondo di attesa per l'apertura del terminale.

	Ciclo iterativo		Chiamate a funzione		Ricorsione	
Numero di messaggi	Tempo totale (s)	Tempo netto (s)	Tempo totale (s)	Tempo netto (s)	Tempo totale (s)	Tempo netto (s)
10	1.563	0.479	1.531	0.461	3.168	0.099
50	2.629	0.517	1.554	0.484	3.166	0.102
100	1.539	0.467	1.654	0.495	5.192	0.104
1000	2.659	0.544	1.726	0.581	1.264	0.247
10000	47.178	1.278	5.542	1.428	18.718	13.194

Tabella 2: Tempo necessario per l'invio di un numero crescente di messaggi

Come si può vedere dalla tabella 2 e in figura 2 (si noti che gli assi sono in scala logaritmica), l'implementazione con la chiamata a funzione è quella mediamente più veloce, anche se ha un tempo netto leggermente maggiore dell'implementazione puramente ricorsiva, anche se resta comparabile.

Una possibile spiegazione per questa peculiarità è che l'invio di messaggi è un processo molto IO-bound, quindi l'approccio iterativo è sufficientemente veloce da non trovare le risorse disponibili e viene sospeso aspettando un interrupt dal sistema operativo, incorrendo in numerosi context switch costosi. La chiamata

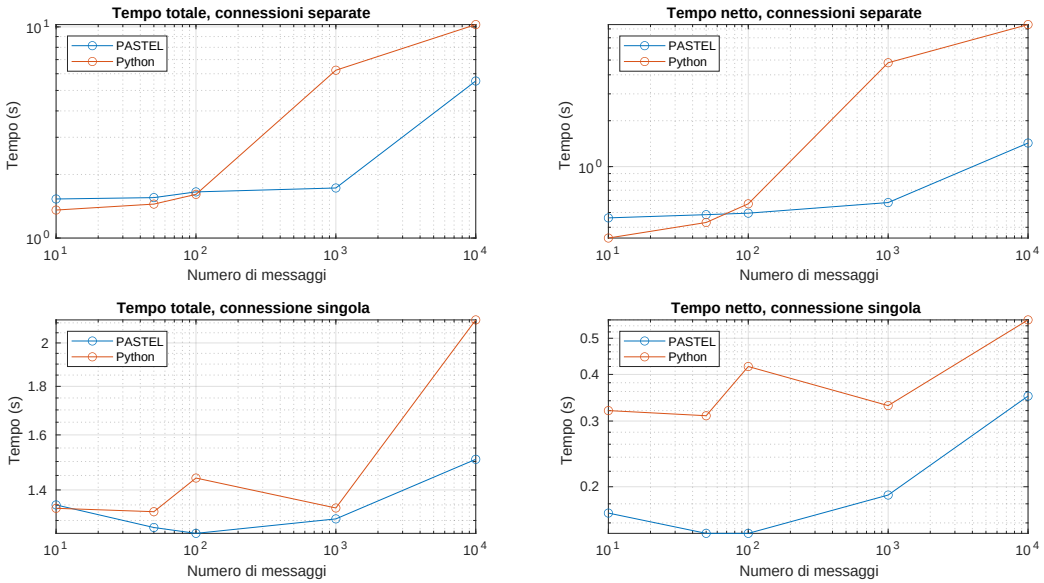

Figura 3: Confronto delle performance di PASTEL e Python

a funzione, invece, ha dell'overhead per la creazione dello stack, questo, come si vede dal tempo netto, è abbastanza piccolo, ma potrebbe essere sufficientemente grande da evitare il context switch.

Questo dato sembra anche giustificare la scelta di sviluppare un linguaggio di scripting interpretato invece che un linguaggio compilato, visto che l'ambito di applicazione principale sarà legato all'invio di messaggi nella rete, e non elaborate manipolazioni dei dati.

Il tempo totale dell'implementazione ricorsiva è il maggiore di tutti[1], nonostante il numero di scope da creare sia lo stesso delle chiamate a funzione iterative, infatti, le ricerche dei simboli nello stack delle chiamate a funzione diventano progressivamente più costose all'aumento delle chiamate ricorsive.

	PASTEL		Python singola connessione		Python connessioni multiple	
Numero di messaggi	Tempo totale (s)	Tempo netto (s)	Tempo totale (s)	Tempo netto (s)	Tempo totale (s)	Tempo netto (s)
10	1.350	0.17	1.339	0.32	1.357	0.340
50	1.278	0.15	1.328	0.31	1.448	0.430
100	1.260	0.15	1.441	0.42	1.610	0.570
1000	1.305	0.19	1.340	0.33	6.232	4.800
10000	1.508	0.35	2.115	0.56	10.254	8.54

Tabella 3: Tempo necessario per l'invio di un numero crescente di messaggi in Python, e in PASTEL con connessione singola

Per potere meglio valutare le performance del linguaggio, è stato deciso di confrontare le prestazioni sullo stesso benchmark con il Python. È stata scelta la modalità "chiamata a funzione", inoltre lo stesso test è stato ripetuto anche con l'invio di più messaggi una singola connessione per valutare una più ampia gamma di scenari. I nuovi risultati sperimentali sono sintetizzati nella tabella 3.

Come si può notare anche dalla figura 3, il programma PASTEL ha sempre prestazioni comparabili se non migliori di quelle del programma equivalente in Python, a testimonianza della validità dell'implementazione.

Tutto il codice utilizzato per i benchmarking è riportato nell'appendice.

[1]Su 10000 messaggi l'esecuzione del programma ricorsivo è stata interrotta prematuramente dall'esaurimento della memoria a disposizione

5 Conclusioni

La progettazione del linguaggio PASTEL ha permesso di affrontare le tematiche trattate nel corso di *Linguaggi e traduttori* e di applicare molti dei concetti teorici discussi durante le lezioni. Nonostante i tempi di sviluppo ridotti, i test effettuati mostrano performance soddisfacenti. Gli strumenti Flex e Bison hanno agevolato la realizzazione di un linguaggio relativamente complesso e completo, riducendo la mole di codice da dovere sviluppare. Visti i tempi di realizzazione limitati, PASTEL non possiede tutte le funzionalità che ci si aspetta da un linguaggio di programmazione general purpose, e sono presenti molte direzioni verso cui si potrebbe potenziare il linguaggio. In conclusione segue una breve disamina di alcune features che potrebbero arricchire il linguaggio.

- Lettura e scrittura da file, per consentire ad esempio la scrittura di log dei device
- Al momento per potere effettuare interazioni con l'utente, quando un programma viene eseguito leggendolo da un file, è necessario ricorrere a `console`. Sarebbe certamente più agevole consentire l'utilizzo dello standard input.
- PASTEL non è utilizzabile su sistemi operativi Windows perché, interfacciarsi alle socket, Windows utilizza un'API differente da Linux e macOS. Un linguaggio maturo dovrebbe avere implementazioni per tutti i maggiori sistemi operativi.
- Gli indirizzi gestiti sono solo quelli IPv4, potrebbe essere aggiunto il supporto al protocollo IPv6.
- Le chiamate a `receive` sono letture bloccanti. Un dispositivo malfunzionante però potrebbe non rispondere mai ai messaggi inviati, fermando l'esecuzione del programma. Per permettere la scrittura di programmi più robusti si potrebbe introdurre letture da socket non bloccanti [20] specificando un timeout.
- Nell'ambito IoT, oltre al TCP, esistono molti protocolli per il trasporto affidabile dei dati basati su UDP [21], supportare anche UDP quindi renderebbe il linguaggio più flessibile e potente.

Riferimenti bibliografici

[1] U. Tandale, B. Momin, and D. P. Seetharam. An empirical study of application layer protocols for iot. In *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, pages 2447–2451, 2017.

[2] Zach Shelby, Klaus Hartke, and Carsten Bormann. The constrained application protocol (coap). 2014.

[3] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Irvine, 2000.

[4] A Banks, E Briggs, K Borgendale, and R Gupta. Mqtt version 5.0. *OASIS Standard*, 2019.

[5] Simon Josefsson et al. The base16, base32, and base64 data encodings. Technical report, RFC 4648, October, 2006.

[6] Salvatore Gaglio, Giuseppe Lo Re, Gloria Martorella, and Daniele Peri. DC4CD: a platform for distributed computing on constrained devices. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(1):1–25, 2017.

[7] Leo Brodie. *Thinking Forth*. Punchy Pub, 2004.

[8] S. Gaglio, G. Lo Re, L. Giuliana, G. Martorella, D. Peri, and A. Montalto. Interoperable real-time symbolic programming for smart environments. In *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 309–316, June 2019.

[9] Don Libes. expect: Scripts for controlling interactive processes. *Computing Systems*, 4(2):99–125, 1991.

[10] Don Libes. expect: Curing those uncontrollable fits of interaction. In *USENIX Summer*, pages 183–192, 1990.

[11] Don Libes. *Exploring Expect: A Tcl-based toolkit for automating interactive programs*. " O'Reilly Media, Inc.", 1995.

[12] Ashok P Nadkarni. *The Tcl Programming Language: A Comprehensive Guide*. CreateSpace Independent Publishing Platform, 2017.

[13] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. A dynamic program analysis to find floating-point accuracy problems. *ACM SIGPLAN Notices*, 47(6):453–462, 2012.

[14] Edsger W Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[15] John Levine. *Flex & Bison: Text Processing Tools*. " O'Reilly Media, Inc.", 2009.

[16] Thomas Narten, Erik Nordmark, William Simpson, and Hesham Soliman. Neighbor discovery for ip version 6 (ipv6), 1998.

[17] Arvind Kandhalu, Karthik Lakshmanan, and Ragunathan Rajkumar. U-connect: a low-latency energy-efficient asynchronous neighbor discovery protocol. In *Proceedings of the 9th ACM/IEEE international conference on information processing in sensor networks*, pages 350–361, 2010.

[18] Ahmad Alsa'deh and Christoph Meinel. Secure neighbor discovery: Review, challenges, perspectives, and recommendations. *IEEE Security & Privacy*, 10(4):26–34, 2012.

[19] time(1) - linux man page.

[20] Brain Beej Hall. Beej's guide to network programming: using internet sockets, 2001.

[21] Madzirin Masirap, Mohd Harith Amaran, Yusnani Mohd Yussoff, Ruhani Ab Rahman, and Habibah Hashim. Evaluation of reliable udp-based transport protocols for internet of things (iot). In *2016 IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE)*, pages 200–205. IEEE, 2016.

Appendice

[b]

Grafico del FSA generato utilizzando GNU Bison 3.5.4 XML Automaton Report. L'immagine è in formato vettoriale, quindi è possibile ingrandire a piacere.

```
def send_messages(dev,times){
int i = 0;
        while(i < times){
                i = i+1;
                connect(dev);
                if(isConnected(dev)){
                        send(dev, toString(i));
                        disconnect(dev);
                } else {
                        print("Couldn't connect to " + toString(dev));
                }
        }
}

device a = localhost:1234;
console(a);
send_messages(a,10000);
quit(0);
```

Listing 5: Codice per inviare messagi su connessioni separate in un loop.

```
def send_message(dev){
        connect(dev);
        if(isConnected(dev)){
                send(dev, toString(i));
                disconnect(dev);
        } else {
                print("Couldn't connect to " + toString(dev));
        }
}

device a = localhost:1234;
console(a);

int i = 0;
while(i < 10000){
        i = i+1;
        send_message(a);
}
quit(0);
```

Listing 6: Codice per inviare messagi su connessioni separate con chiamate a funzione in un loop.

```
def send_message(dev, times){
        connect(dev);
        if(isConnected(dev)){
                send(dev, toString(times)+" ");
                disconnect(dev);
        } else {
                print("Couldn't connect to " + toString(dev));
        }
        if(times){
                send_message(dev, times-1);
        }
```

```
12 }
13
14 device a = localhost:1234;
15 console(a);
16
17 send_message(a,10000);
18 quit(0);
```

Listing 7: Codice per inviare messagi su connessioni separate ricorsivamente.

```
1  import binascii
2  from pwn import *
3  def send(r,num):
4      r.sendline(str(num))
5  port = 1234
6  server = '127.0.0.1'
7  sleep(1)
8  for i in range(10000):
9      r = remote(server, port)
10     send(r,i)
11     r.close()
```

Listing 8: Codice Python per inviare messagi su connessioni separate con chiamate a funzione in un loop.

```
1  def send_message(dev, num){
2      send(dev, toString(num));
3  }
4  device a = localhost:1234;
5  console(a);
6  connect(a);
7  int i = 0;
8  while(i < 10){
9      send_message(a, i);
10     i = i+1;
11 }
12 disconnect(a);
13 quit(0);
```

Listing 9: Codice PASTEL per inviare messagi su una singola connessione con chiamate a funzione in un loop.

```
1  import binascii
2  from pwn import *
3  def send(r,num):
4      r.sendline(str(num))
5  port = 1234
6  server = '127.0.0.1'
7  sleep(1)
8  r = remote(server, port)
9  for i in range(10):
10     send(r,i)
11 r.close()
```

Listing 10: Codice Python per inviare messagi su una singola connessione con chiamate a funzione in un loop.