net2o: Command Language

Eine universelle Sprache für strukturierte Daten und RPC

Bernd Paysan

11. April, Forth–Tagung 2015, Hannover

Übersicht

Kommunikation im Forth–Stil

Anforderungen für sichere Kommunikation (sicher wie in „keine Exploits durch Fehlinterpretation“)

- Extrem einfacher Interpreter
- Erweiterbar, aber der Empfänger muss die Erweiterungen erlauben
- Universell, also nur ein Interpreter, den man auditen und verifizieren muss
- Die triviale Einfachheit macht es schwierig, das zu erklären

Kommunikation im Forth–Stil

Anforderungen für sichere Kommunikation (sicher wie in „keine Exploits durch Fehlinterpretation“)

- Extrem einfacher Interpreter
- Erweiterbar, aber der Empfänger muss die Erweiterungen erlauben
- Universell, also nur ein Interpreter, den man auditen und verifizieren muss
- Die triviale Einfachheit macht es schwierig, das zu erklären

Kommunikation im Forth–Stil

Anforderungen für sichere Kommunikation (sicher wie in „keine Exploits durch Fehlinterpretation“)

- Extrem einfacher Interpreter
- Erweiterbar, aber der Empfänger muss die Erweiterungen erlauben
- Universell, also nur ein Interpreter, den man auditen und verifizieren muss
- Die triviale Einfachheit macht es schwierig, das zu erklären

Kommunikation im Forth–Stil

Anforderungen für sichere Kommunikation (sicher wie in „keine Exploits durch Fehlinterpretation“)

- Extrem einfacher Interpreter
- Erweiterbar, aber der Empfänger muss die Erweiterungen erlauben
- Universell, also nur ein Interpreter, den man auditen und verifizieren muss
- Die triviale Einfachheit macht es schwierig, das zu erklären

Kommunikation im Forth–Stil

Anforderungen für sichere Kommunikation (sicher wie in „keine Exploits durch Fehlinterpretation“)

- Extrem einfacher Interpreter
- Erweiterbar, aber der Empfänger muss die Erweiterungen erlauben
- Universell, also nur ein Interpreter, den man auditen und verifizieren muss
- Die triviale Einfachheit macht es schwierig, das zu erklären

Grundlagen

- Fünf Datentypen: Integer (64 Bit mit und ohne Vorzeichen), Flag, String (generisches Byte–Array), IEEE float, Objekte.
- Instruktionen und Daten werden wie Protobuf[?] codiert (7 bits pro Byte, MSB=1 bedeutet „Daten gehen weiter“, höchstwertiger Teil zuerst)
- Vier Stacks: integer, float, objects, strings
- `endwith` und `endcmd` um Objekt–Message–Blöcke und Kommandos zu beenden
- `oswap` um das aktuelle Objekt für's Einfügen in das Vater–Objekt auf den Objekt–Stack zu legen
- `words` für Reflections (die Wörter werden mit Token–Nummer, Identifier und Stack–Effekt gelistet, damit automatische Bindings erzeugbar sind)

Grundlagen

- Fünf Datentypen: Integer (64 Bit mit und ohne Vorzeichen), Flag, String (generisches Byte–Array), IEEE float, Objekte.
- Instruktionen und Daten werden wie Protobuf[**?**] codiert (7 bits pro Byte, MSB=1 bedeutet „Daten gehen weiter“, höchstwertiger Teil zuerst)
- Vier Stacks: integer, float, objects, strings
- `endwith` und `endcmd` um Objekt–Message–Blöcke und Kommandos zu beenden
- `oswap` um das aktuelle Objekt für's Einfügen in das Vater–Objekt auf den Objekt–Stack zu legen
- `words` für Reflections (die Wörter werden mit Token–Nummer, Identifier und Stack–Effekt gelistet, damit automatische Bindings erzeugbar sind)

Grundlagen

- Fünf Datentypen: Integer (64 Bit mit und ohne Vorzeichen), Flag, String (generisches Byte–Array), IEEE float, Objekte.
- Instruktionen und Daten werden wie Protobuf[**?**] codiert (7 bits pro Byte, MSB=1 bedeutet „Daten gehen weiter“, höchstwertiger Teil zuerst)
- Vier Stacks: integer, float, objects, strings
- `endwith` und `endcmd` um Objekt–Message–Blöcke und Kommandos zu beenden
- `oswap` um das aktuelle Objekt für's Einfügen in das Vater–Objekt auf den Objekt–Stack zu legen
- `words` für Reflections (die Wörter werden mit Token–Nummer, Identifier und Stack–Effekt gelistet, damit automatische Bindings erzeugbar sind)

Grundlagen

- Fünf Datentypen: Integer (64 Bit mit und ohne Vorzeichen), Flag, String (generisches Byte–Array), IEEE float, Objekte.
- Instruktionen und Daten werden wie Protobuf[**?**] codiert (7 bits pro Byte, MSB=1 bedeutet „Daten gehen weiter“, höchstwertiger Teil zuerst)
- Vier Stacks: integer, float, objects, strings
- `endwith` und `endcmd` um Objekt–Message–Blöcke und Kommandos zu beenden
- `oswap` um das aktuelle Objekt für's Einfügen in das Vater–Objekt auf den Objekt–Stack zu legen
- `words` für Reflections (die Wörter werden mit Token–Nummer, Identifier und Stack–Effekt gelistet, damit automatische Bindings erzeugbar sind)

Grundlagen

- Fünf Datentypen: Integer (64 Bit mit und ohne Vorzeichen), Flag, String (generisches Byte–Array), IEEE float, Objekte.
- Instruktionen und Daten werden wie Protobuf[**?**] codiert (7 bits pro Byte, MSB=1 bedeutet „Daten gehen weiter“, höchstwertiger Teil zuerst)
- Vier Stacks: integer, float, objects, strings
- `endwith` und `endcmd` um Objekt–Message–Blöcke und Kommandos zu beenden
- `oswap` um das aktuelle Objekt für's Einfügen in das Vater–Objekt auf den Objekt–Stack zu legen
- `words` für Reflections (die Wörter werden mit Token–Nummer, Identifier und Stack–Effekt gelistet, damit automatische Bindings erzeugbar sind)

Grundlagen

- Fünf Datentypen: Integer (64 Bit mit und ohne Vorzeichen), Flag, String (generisches Byte–Array), IEEE float, Objekte.
- Instruktionen und Daten werden wie Protobuf[**?**] codiert (7 bits pro Byte, MSB=1 bedeutet „Daten gehen weiter“, höchstwertiger Teil zuerst)
- Vier Stacks: integer, float, objects, strings
- `endwith` und `endcmd` um Objekt–Message–Blöcke und Kommandos zu beenden
- `oswap` um das aktuelle Objekt für's Einfügen in das Vater–Objekt auf den Objekt–Stack zu legen
- `words` für Reflections (die Wörter werden mit Token–Nummer, Identifier und Stack–Effekt gelistet, damit automatische Bindings erzeugbar sind)

Warum binär codieren?

- Schneller und einfacher zu parsen (einfacher bedeutet kleinere Angriffsfläche)
- Es gibt immer noch die Möglichkeit, Kommandos über ein Frontend als Klartext einzugeben
- Der Detokenizer für's Debuggen ist auch sehr einfach
- Der objekt–orientierte Ansatz macht das Schreiben von Anwendungsspezifischer Logik sehr einfach

Warum binär codieren?

- Schneller und einfacher zu parsen (einfacher bedeutet kleinere Angriffsfläche)
- Es gibt immer noch die Möglichkeit, Kommandos über ein Frontend als Klartext einzugeben
- Der Detokenizer für's Debuggen ist auch sehr einfach
- Der objekt–orientierte Ansatz macht das Schreiben von Anwendungsspezifischer Logik sehr einfach

Warum binär codieren?

- Schneller und einfacher zu parsen (einfacher bedeutet kleinere Angriffsfläche)
- Es gibt immer noch die Möglichkeit, Kommandos über ein Frontend als Klartext einzugeben
- Der Detokenizer für's Debuggen ist auch sehr einfach
- Der objekt–orientierte Ansatz macht das Schreiben von Anwendungsspezifischer Logik sehr einfach

Warum binär codieren?

- Schneller und einfacher zu parsen (einfacher bedeutet kleinere Angriffsfläche)
- Es gibt immer noch die Möglichkeit, Kommandos über ein Frontend als Klartext einzugeben
- Der Detokenizer für's Debuggen ist auch sehr einfach
- Der objekt–orientierte Ansatz macht das Schreiben von Anwendungsspezifischer Logik sehr einfach

Warum eine Programmiersprache als Daten?

Lemma: jede glue logic wird irgendwann Turing–complete

- Implementiere nur das, was du brauchst — aber du solltest nicht mehr als einen generischen Interpreter implementieren
- Die typische Vorgehensweise für RPC: Das ganze Objekt serialisieren (mit Unterobjekten), über's Netz senden, und eine Funktion auf das Objekt anwenden
- Das Konzept von Net2o (basiert auf ONF): Halte das Objekt synchron und sende nur die Änderungen — mit einfachen Settern und Gettern, und was man sonst für Funktionen braucht
- Damit kann man viele Aufrufe auf einmal übertragen, und reduziert die Latenz

Warum eine Programmiersprache als Daten?

Lemma: jede glue logic wird irgendwann Turing–complete

- Implementiere nur das, was du brauchst — aber du solltest nicht mehr als einen generischen Interpreter implementieren
- Die typische Vorgehensweise für RPC: Das ganze Objekt serialisieren (mit Unterobjekten), über's Netz senden, und eine Funktion auf das Objekt anwenden
- Das Konzept von Net2o (basiert auf ONF): Halte das Objekt synchron und sende nur die Änderungen — mit einfachen Settern und Gettern, und was man sonst für Funktionen braucht
- Damit kann man viele Aufrufe auf einmal übertragen, und reduziert die Latenz

Warum eine Programmiersprache als Daten?

Lemma: jede glue logic wird irgendwann Turing–complete

- Implementiere nur das, was du brauchst — aber du solltest nicht mehr als einen generischen Interpreter implementieren
- Die typische Vorgehensweise für RPC: Das ganze Objekt serialisieren (mit Unterobjekten), über's Netz senden, und eine Funktion auf das Objekt anwenden
- Das Konzept von Net2o (basiert auf ONF): Halte das Objekt synchron und sende nur die Änderungen — mit einfachen Settern und Gettern, und was man sonst für Funktionen braucht
- Damit kann man viele Aufrufe auf einmal übertragen, und reduziert die Latenz

Warum eine Programmiersprache als Daten?

Lemma: jede glue logic wird irgendwann Turing–complete

- Implementiere nur das, was du brauchst — aber du solltest nicht mehr als einen generischen Interpreter implementieren
- Die typische Vorgehensweise für RPC: Das ganze Objekt serialisieren (mit Unterobjekten), über's Netz senden, und eine Funktion auf das Objekt anwenden
- Das Konzept von Net2o (basiert auf ONF): Halte das Objekt synchron und sende nur die Änderungen — mit einfachen Settern und Gettern, und was man sonst für Funktionen braucht
- Damit kann man viele Aufrufe auf einmal übertragen, und reduziert die Latenz

Sicherheit

Lemma: jedes hinreichend komplexe Format kann missbraucht werden

Also beschränkt man sich auf ein ganz einfaches Format

Interpreter

```
: cmd@ ( -- u )
  buf-state 2@ over + >r p@+ r> over - buf-state 2! 64>n ;
: n>cmd ( n -- addr )  cells >r
  o IF   token-table  ELSE  setup-table  THEN
  $@ r@ u<= IF  net2o-crash  THEN  r> + ;
: cmd-dispatch ( addr u -- addr' u' )  buf-state 2!
  cmd@ n>cmd @ ?dup IF  execute  ELSE  net2o-crash  THEN
  buf-state 2@ ;
: cmd-loop ( addr u -- )
  BEGIN  cmd-dispatch dup 0<= UNTIL  2drop ;
```

Sicherheit

Lemma: jedes hinreichend komplexe Format kann missbraucht werden

Also beschränkt man sich auf ein ganz einfaches Format

Interpreter

```
: cmd@ ( -- u )
  buf-state 2@ over + >r p@+ r> over - buf-state 2! 64>n ;
: n>cmd ( n -- addr )  cells >r
  o IF   token-table  ELSE  setup-table  THEN
  $@ r@ u<= IF  net2o-crash  THEN  r> + ;
: cmd-dispatch ( addr u -- addr' u' )  buf-state 2!
  cmd@ n>cmd @ ?dup IF  execute  ELSE  net2o-crash  THEN
  buf-state 2@ ;
: cmd-loop ( addr u -- )
  BEGIN  cmd-dispatch dup 0<= UNTIL  2drop ;
```

Dateien lesen

lese drei Dateien

```
0 lit, file-id "net2o.fs" $, 0 lit,
open-file get-size get-stat endwith
1 lit, file-id "data/2011-05-13_11-26-57-small.jpg" $, 0 lit,
open-file get-size get-stat endwith
2 lit, file-id "data/2011-05-20_17-01-12-small.jpg" $, 0 lit,
open-file get-size get-stat endwith
```

Dateien lesen: Antwort

lese drei Dateien: Antwort

```
0 lit, file-id 12B9A lit, set-size
    2014-8-24T13:52:27.220Z lit, 1A4 lit, set-stat endwith
1 lit, file-id 9C65C lit, set-size
    2014-7-27T00:34:15.309Z lit, 1A4 lit, set-stat endwith
2 lit, file-id 9D240 lit, set-size
    2014-7-27T00:34:15.427Z lit, 1A4 lit, set-stat endwith
```

Messages

messages

```
msg [: msg-start "Hi Bernd" $, msg-text ;]+
    "<pubkey>"+"<date-span>"+"<signature>" $,
    nestsig endwith
"<reply-token>" push-$ push' nest 0 ok?
```

Structured Text a la HTML

HTML–like structured text

```
body
   p "Some text with " text
       bold "bold" text oswap add
       " markup" text
   oswap add
   li
       ul "a bullet point" text oswap add
       ul "another bullet point" text oswap add
   oswap add
oswap add
```

Literatur&Links I

Bernd Paysan
net2o source repository und Wiki
http://fossil.net2o.de/net2o

Google Developers
Protocol Buffers
https://developers.google.com/protocol-buffers/