Rust Process Networks

Ian Daniher - Feb 17 2014

(Kahn) Process Networks

Formalizable mapping from (infinite) input sequences to (infinite) output sequences.

“Sends” complete “instantly.”

“Receives” wait until data is available.

1974 paper by GH Kahn.

KPNs: Wat.

We want to use computers to process streams of real-world information, like radio signals, sound, or sensor data. This is harder than it sounds.

Process networks make it a little easier.

Rust makes it really easy.

KPNs: Useful for DSP

“Real-time Sonar Beamforming on a Unix Workstation Using Process Networks and POSIX Threads” (1998)

KPNs: Simple.

KPNs match Rust task / stream semantics.

Restricted in comparison to Actor (Erlang) or CSP (Go) models.

Blocking reads provide synchronization.

Data-driven execution.

(Rust) Process Networks

“a safe, concurrent, practical language”

- lightweight processes
- nonblocking sends, blocking recvs
- fast!

Why bother?

GNURadio

Simulink

Labview

Max/MSP

Why bother?

Heavy.

or Proprietary.

or Both!

Why bother?

ADE: Affordable Design and Entrepreneurship.

$100 COTS BOM to provide 4x4 “real world” IO

Grow plants with computers! Replace black boxes with software! Make more food!

How?

Rust is statically typed.

Rust is new.

Rust is low level.

How?

Recursion!

Pluggable Blocks!

How?

KPNs deal with (infinite) sequences of “Tokens.”

Great, let’s make a “Token” type!

How?

KPNs deal with (infinite) sequences of “Tokens.”

```
pub enum Token {
        Chip(uint),
        Dbl(f64),
        Break(~str),
        Dur(~Token, f64),
        Run(~Token, uint),
        Packet(~[Token]),
}
```

How?

```
pub enum Token {
        Chip(uint),
        Dbl(f64),
        Break(~str),
        Dur(~Token, f64),
        Run(~Token, uint),
        Packet(~[Token]),
}
```

basic symbol

basic number

terminator

duration

sequence

recursive store

How?

```
pub fn printdump(U: Port<Token>, S: SourceConf) {
        loop {
                match U.recv() {
                        Packet(x) => println!("{:?}", (x.
len(), x)),
                        x => println!("{:?}", x),
                }
        }
}
```

Tokens can be inspected and consumed.

How?

```
pub enum Parts{
        Head (fn (Chan<Token>, SourceConf) -> () ),
        Body (fn (Port<Token>, Chan<Token>,
SourceConf) -> () ),
        Tail (fn (Port<Token>, SourceConf) -> () ),
        Fork (fn (Port<Token>, Chan<Token>,
Chan<Token>) -> () ),
        Funnel (fn (Port<Token>, Port<Token>,
Chan<Token>) -> ()),
        Leg (~[Parts] ),
}
```

Streams of tokens can be sourced, consumed, processed, duplicated, combined, and abstracted.

How?

Streams of tokens can be sourced, consumed, processed, duplicated, combined, and abstracted.

These operations can be structured as a directed acyclical* graph.

*for now

How?

These operations can be structured as a directed acyclical* graph.

*for now

Ideally, we could represent a flowgraph and merely pass it to a function that would instantiate it.

How?

```
fn main() {
    let conf = SourceConf{Freq: 434e6, Rate: 1.024e6, Period: 5e-4};
    let t1 = ~[Body(sensors::validTokenA), Body(pkt36), Body(pktTempA), Body(sensors::sensorUnpackerA)];
    let t2 = ~[Body(kpn::validTokenManchester), Body(kpn::manchesterd), Body(pkt195),
            Body(pktTempB), Body(sensors::sensorUnpackerB)];
    let parsing: ~[Parts] = ~[Body(bitfount::discretize), Body(kpn::rle), Body(kpn::dle), Fork(kpn::tuplicator), Leg
(t2),
        Funnel(kpn::twofunnel), Tail(kpn::udpTokenSink)];
    let fs: ~[Parts] = ~[Head(bitfount::rtlSource), Body(bitfount::trigger), Fork(kpn::tuplicator),
        Leg(~[Tail(vidsink::vidSink)]), Leg(parsing)];
    instant::spinUp(fs, ~[], conf);
    loop {}
}
```

Yeah.

libinstant is cool.

~80 lines of code that walks a naive DAG process network representation and instantiates it, backed with the linux scheduler and rust streams.

Pluggable.

Certainly not going to write a frontend with SDL.

LibRedio needs to talk with “stuff.”

Pluggable.

“MessagePack is an efficient binary serialization format. It lets you exchange data among multiple languages like JSON. But it's faster and smaller.”

UDP (User Datagram Protocol) is about as simple as networking gets. And works great for distributing tokens to other processes on the same computer.

Pluggable.

The previously illustrated Token type maps to msgpack flawlessly.

```
pub fn tokenToValue(U: Token) -> Value {
        match U {
                Packet(p) => Array(p.move_iter().map(|x| tokenToValue(x)).to_owned_vec()),
                Dbl(x) => Double(x),
                Chip(x) => Unsigned(x as u64),
                Break(s) => String(s.into_bytes()),
                Dur(~t,d) => Array(~[tokenToValue(t), tokenToValue(Dbl(d))]),
                Run(~t,d) => Array(~[tokenToValue(t), tokenToValue(Chip(d))]),
        }
}
```

What Next?

Feedback!

More feedback!

More hardware support!

What Next?

Questions? Comments?