The case for reactive objects

Johan Nordlander, Luleå Univ. och Technology

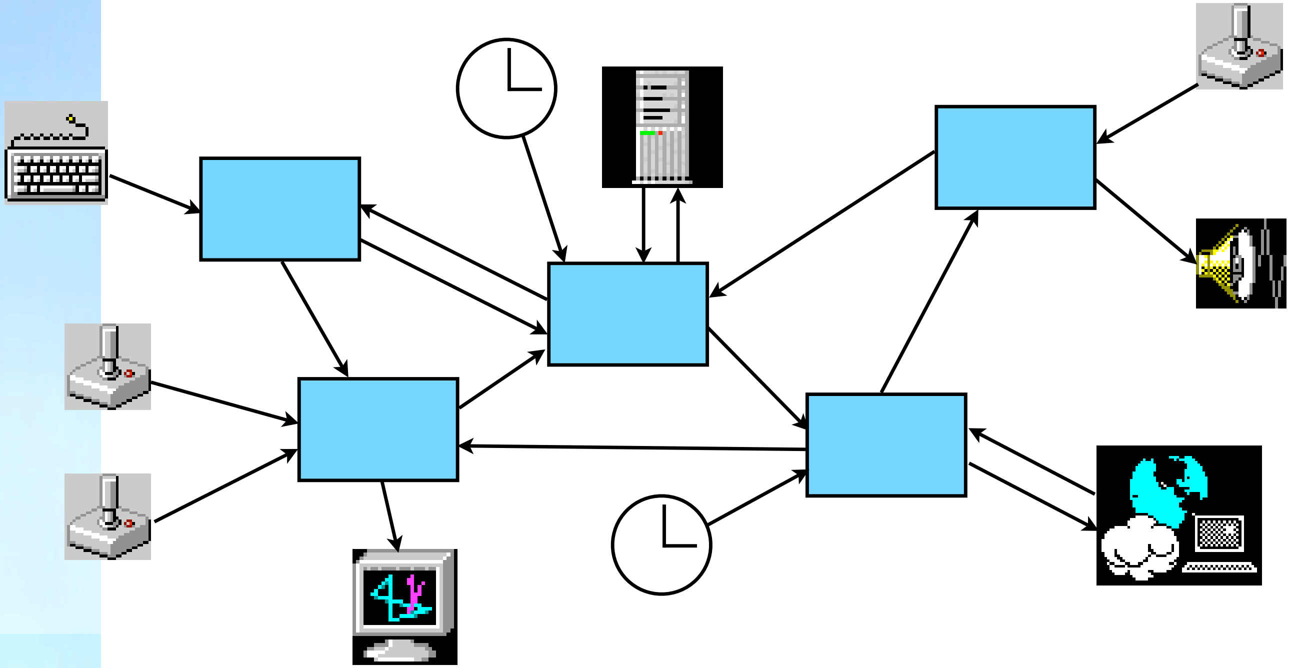
(with Mark Jones, Andrew Black, Magnus Carlsson, Dick Kieburtz – all (ex) OGI)

Links meeting, April 6

Links killer apps

- Web services...
- Games...
- Web-based games...

A challenge to implement!



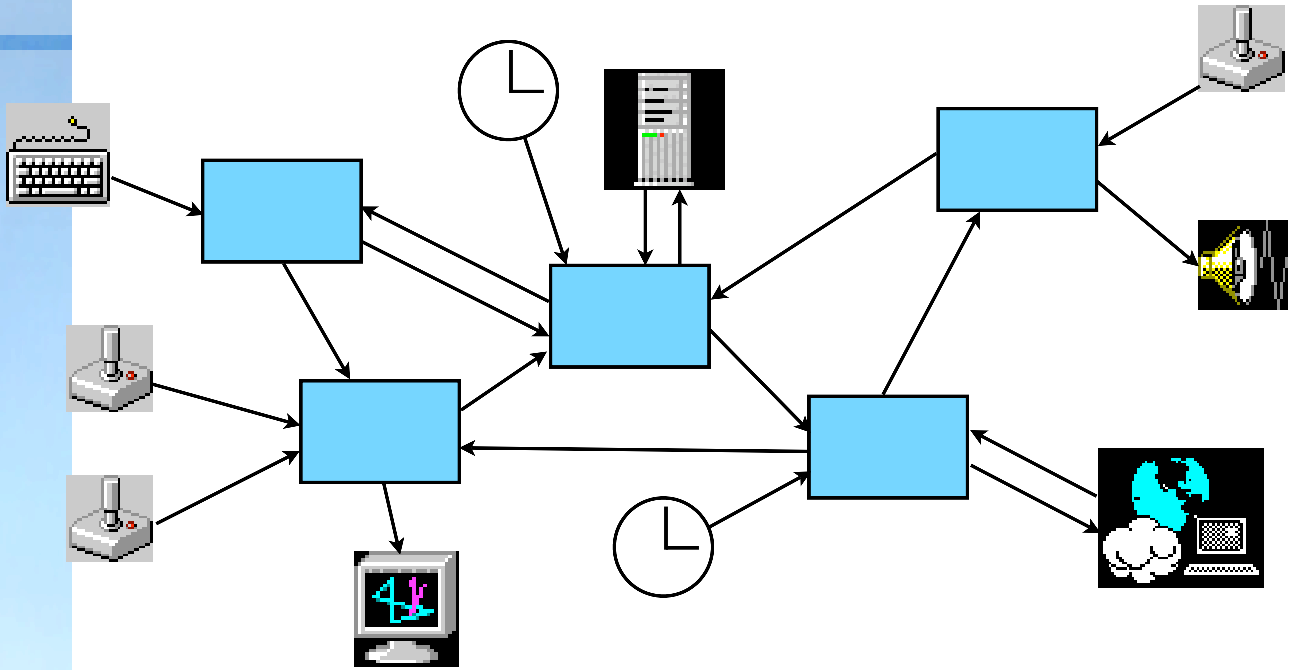
Particular challenges

- Multiple, asynchronous inputs
  - Languages tend to allow only one input at a time (read symmetric to write)
- Distributed state and concurrency
  - Languages tend to decouple state from concurrency
    - OO languages structure according to state, concurrency aspect crosscuts the OO design
    - Concurrent languages structure around threads, shared state must be manually protected

Erlang

- Supports blocking for multiple messages
- Lets state follow a process
- However, Erlang is
  - untyped
  - not referentially transparent
  - still dependent on encodings, in order to support a model of communicating boxes
    - event-loop pattern
    - restricted use of the blocking op receive
    - disciplined use of message tags

Back to our boxes

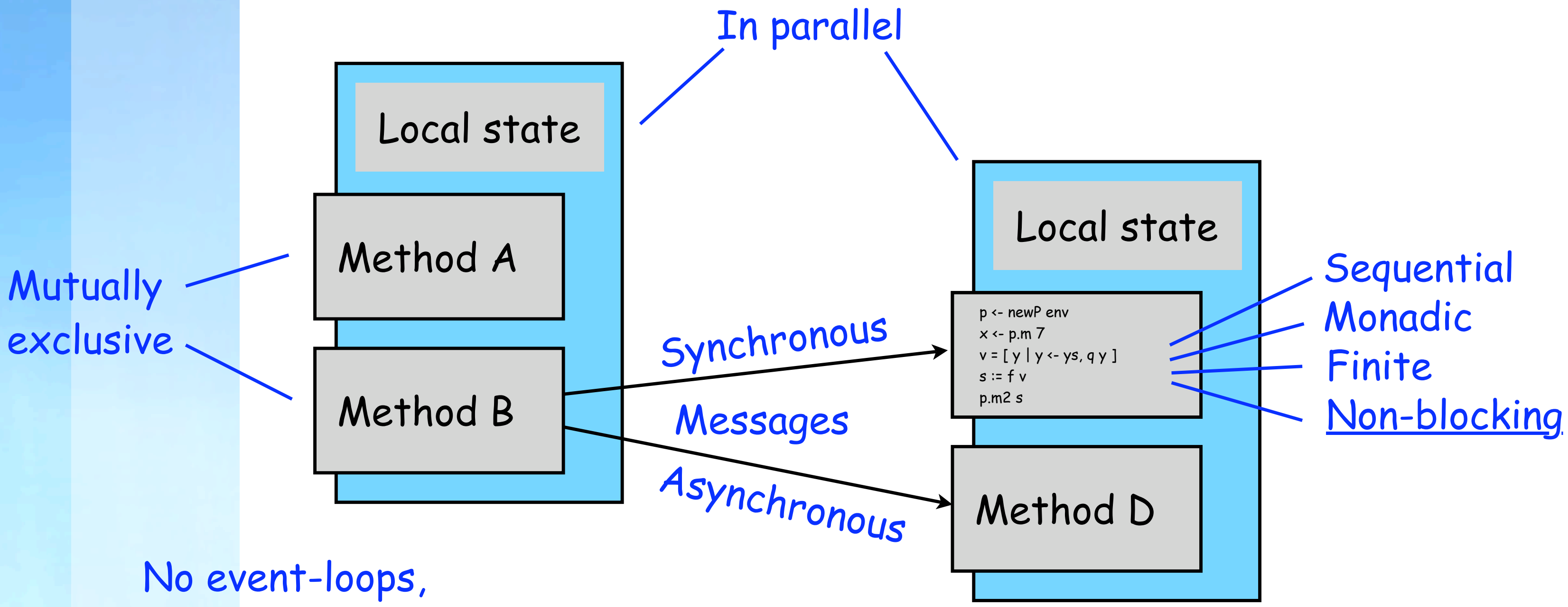


- Notice the OO intuition!
- What stops languages from directly supporting boxes that are both objects (encapsulating a state, communicating with messages) and processes (evolving in parallel)?

Timber...

- ... is such a language (an evolution of O'Haskell, which in turn is an OO and concurrency extension of Haskell)

http://www.csee.ltu.se/index.php?subject=timber



In parallel

Local state

Method A

Method B

Mutually exclusive

Synchronous

Messages

Asynchronous

Local state

```
p <- newP env
x <- p.m 7
v = [ y | y <- ys, q y ]
s := f v
p.m2 s
```

Method D

Sequential
Monadic
Finite
Non-blocking

No event-loops, objects are passive by default

The role of objects

- Core programming model:

  Every object is a process

- Equally important:

  Everything is not an object!

- Values (lists, trees, records, functions, ...) replace most uses of objects in traditional OO
- Timber objects correspond closely to Erlang processes (including efficiency implications)
- Timber is strict, and purely functional (in the Haskell sense), with a stratified formal semantics (λ+CHAM)
- Also first-class: methods (important for callbacks)

Example

- A directory server:

```
directoryServer =
    template
        assoc := []
        insert k v = action
            assoc := (k,v) : assoc
        query k = request
            return (lookup k assoc)
        return (Directory {...})
```

- Using it:

```
s <- directoryServer
...
s.insert "Johan" 12345
...
v <- s.query "Johan"
```

In Erlang:

```
serverloop(Assoc) ->
    receive
        {insert, K, V} ->
            serverloop([{K,V}|Assoc]);
        {query, K, Pid} ->
            Pid ! {reply,lookup(K,Assoc)},
            serverloop(Assoc)
    end.

S = spawn(fun()->serverloop([]) end),
...
S ! {insert, "Johan", 12345},
...
S ! {query, "Johan", self()},
receive  {reply,V} -> ...  end
```

Types

- Message-passing = calling methods
- Object/process interfaces can thus be described as a <u>product of methods</u> (c.f. using channels and sum types):

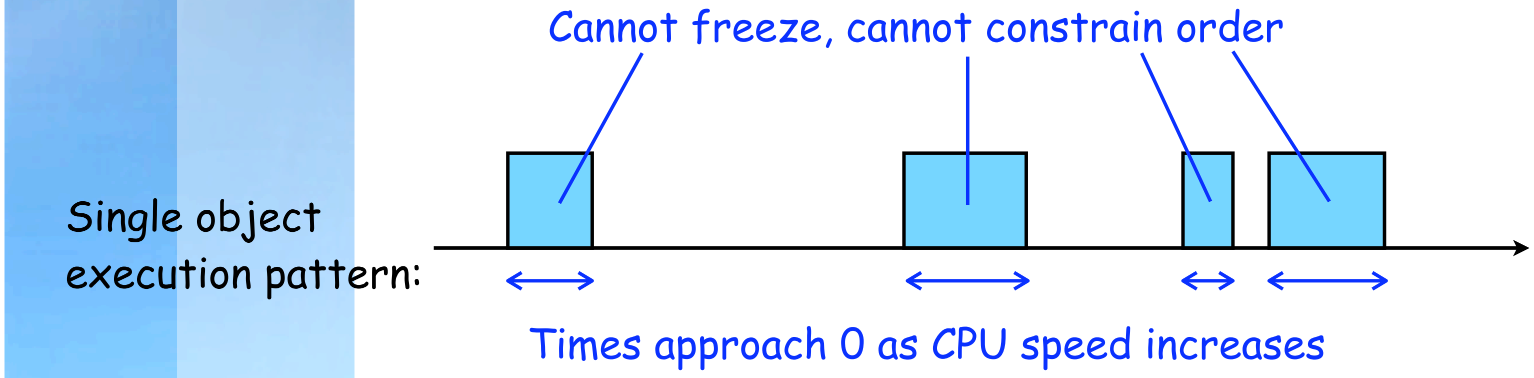
```
struct Directory a =
    insert :: Key -> a -> Action                asynchronous method
    query  :: Key -> Request (Maybe a)          synchronous method
```

- Note that communication semantics, including rendezvous result, is visible in types
- Unreliable communication can also be captured:

```
unreliable_query :: Key -> (Maybe a -> Action) -> Action
```

- In general, object interfaces can be <u>any data structure</u> containing methods, and a single object can support <u>multiple interfaces</u>

Reactivity


Cannot freeze, cannot constrain order
Single object execution pattern:
Times approach 0 as CPU speed increases

- Objects are “always” responsive
- Events unify with method calls (never with returns)
- Decentralized event-handling by every object
- Close to the plain communicating-boxes-model (no stuck states that transparently hook up clients)

More

- Components:

  ```
  comp1 :: A -> Template B
  comp2 :: B -> Template C
  comp3 = comp1 <||> comp2
  comp3 :: A -> Template C
  ```

  - Declare object generators, not objects directly (stateless source code)
  - No global interfaces, object dependencies through parameters only
- Nominal subtyping system, integrated in qualified types framework
- Upper and lower time-constraints on methods (time-driven behavior and deadline scheduling)

Last slide

- Reactive objects (à la Timber) offers:
  - event handling and concurrency, with enforced
    - state encapsulation
    - state protection (mutual exclusion)
    - responsivity
  - object-orientation (not in the Java sense, but in the classical modelling sense)
  - type-safe communication with precise interfaces
  - a matching context for purely functional programming
- Would any of that fit into Links?