Task Oriented Programming
with

-

A Domain Specific Language embedded in

Rinus Plasmeijer – Bas Lijnse
Peter Achten – Pieter Koopman - Steffen Michels
Jurriën Stutterheim - Markus Klinik - Tim Steenvoorden - Mart Lubbers – Arjan Oortgiesen
Jan Martin Jansen (NLDA) – John van Groningen - Laszlo Domoszlai (ELTE)
Radboud University Nijmegen

Overview

- Introduction to Task Oriented Programming
- iTask Overview
  - Task Values
  - Editors
  - Task Combinators
    - Sequential Combinators
    - Parallel Combinators
  - Shared Data
  - Current Research

Work in Progress & Future Work

- Implementation
  - Deployement / Efficiency / Polishing / Library Extensions of the iTask System (Bas Lijnse)
  - Code Generation (John van Groningen)
  - Interpreter (Camil Straps, Erin van der Veen)
  - (Graphical) Editors (Lucas Franceschino, Peter Achten, Bas Lijnse)

  - Distributed Version / Android Apps (Haye Bohm, Arjan Oortgiese)
  - Integrate with Internet of Things (Mart Lubbers, Pieter Koopman, Matheus Amazonas)
  - IoT costs analysis (Erin van der Veen)

  - Security issues (Mark Wijkhuizen)

- Semantic Issues

  - Resource Analysis (Markus Klinik)
  - What is the Semantics, what are properties (Sjaak Smetsers, Lucas Frenceschino,Tim, Markus, Nico Naus)

- End Users

  - Graphical Feedback et compile time & run-time (Jurrien Stutterheim)
  - Graphical Development Tool for non-experts (Tim Steenvoorden)
  - Hint Feedback to end users:
    - what are the best moves to make progress (Nico Naus)

- Applications

A
Distributed Dynamic Architecture
for
itasks
dynamic workflow system

-

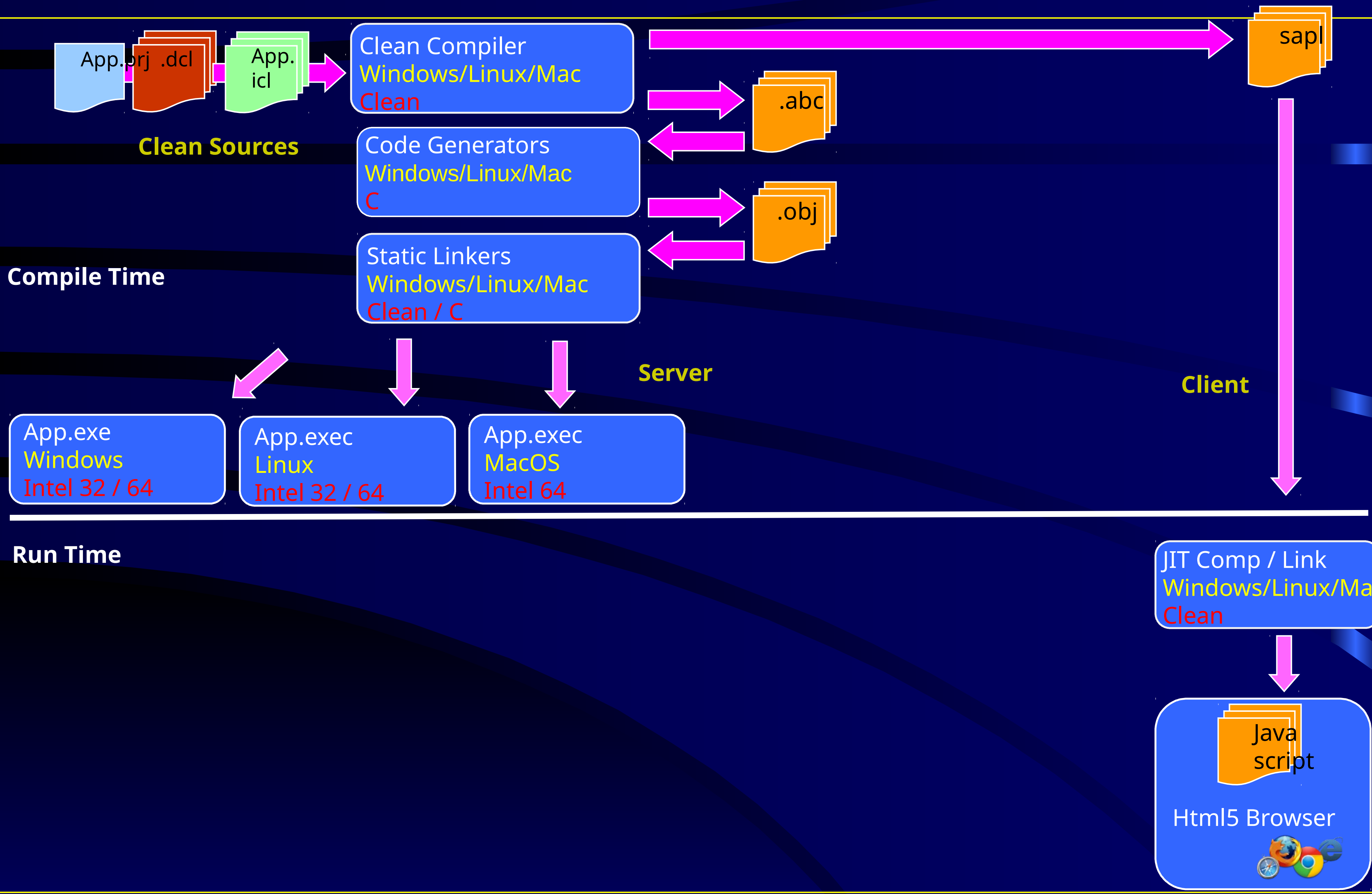
Arjen Oortgiese

Peter Achten – John van Groningen

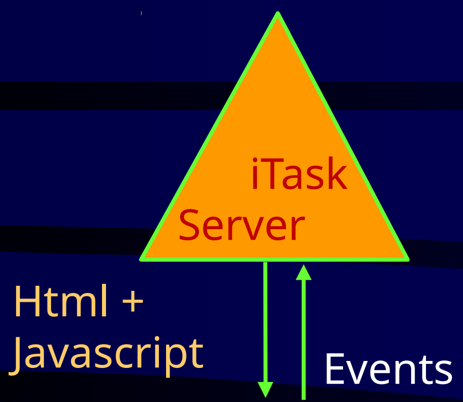
Rinus Plasmeijer

Radboud University Nijmegen

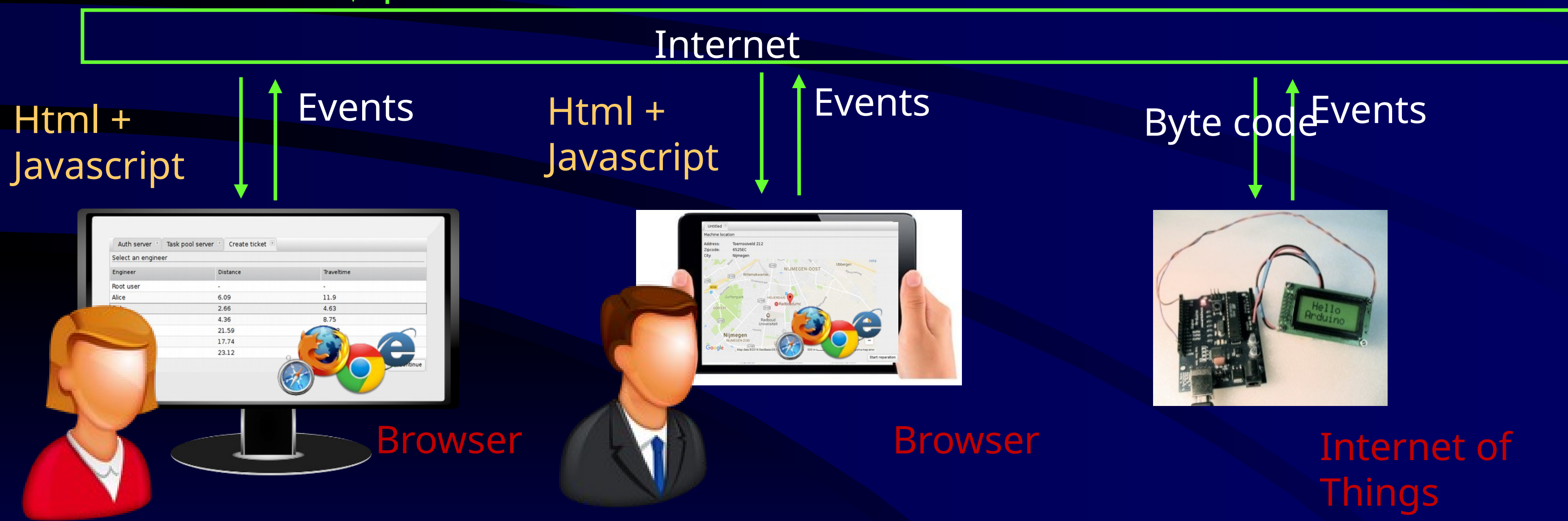
1 Source Solution – Compiled <u>twice</u> for Server & Clients



Standard iTask Architecture: 1 Server – browser Clients



- Login Adminstration (end users)
- Task Administration (which tasks need to be done by whom)
- Task Instance Administration
  - Coordination of tasks
  - Who is working on what, what has to be done next
  - What are the consequences when a task value changes
- Share Administration
  - What are the consequences when an SDS value changes



Advantages / Disadvantages standard 1 Server Solution

+ Advantages: Relatively simple architecture, works fine

- Disadvantages: Server Not scalable : server too busy when too many clients login

  Runs on Intel based platforms (Linux, Mac, Windows) only

  Clients Need the server: One cannot work offline

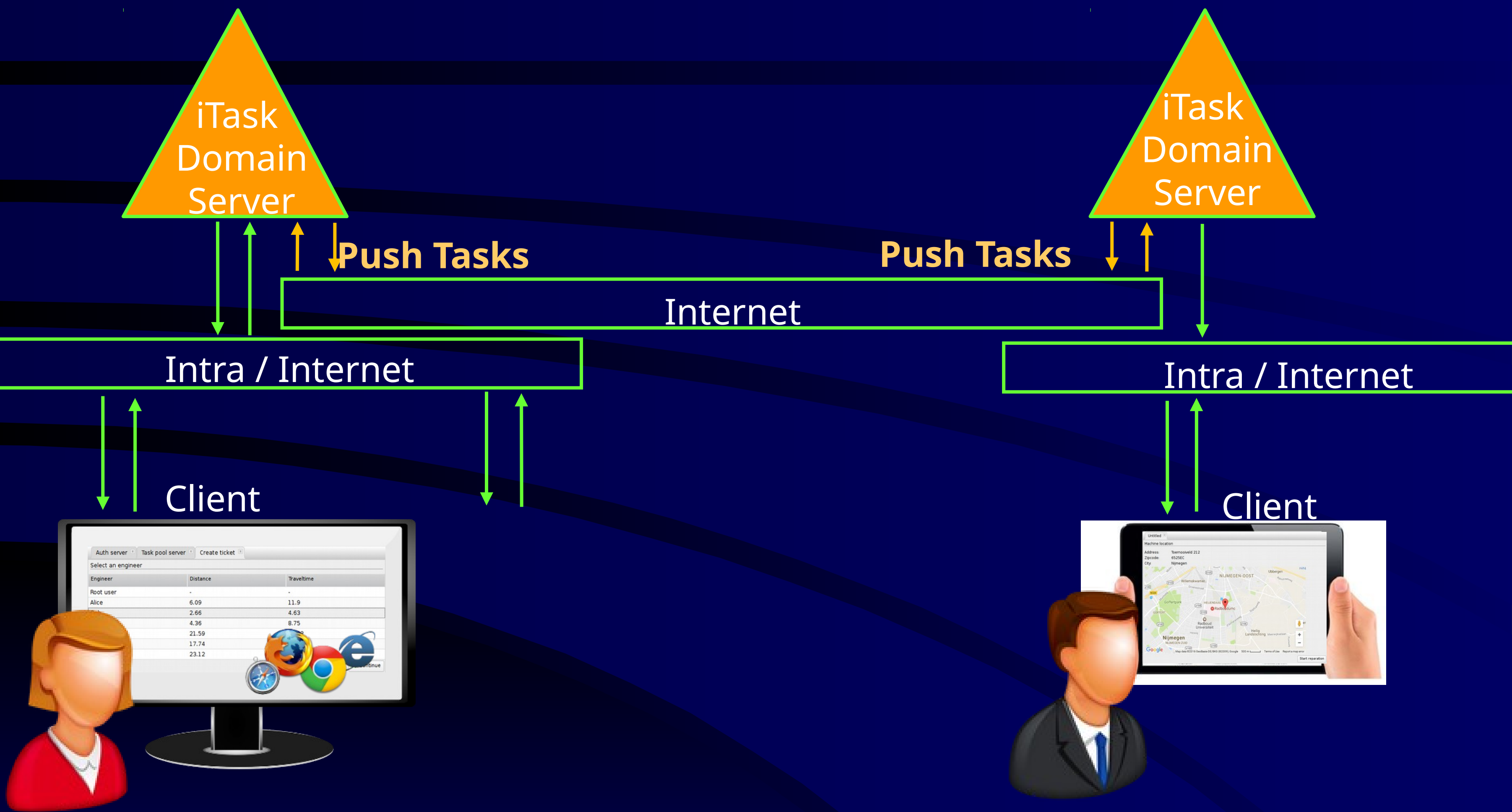
  Browser limitations: Limited access to hardware of e.g. a mobile phone

  Javascript: Very Slow (Clean is about 10 times faster)

Distributed iTask Domain Servers

- Distribute the tasks over Domain Servers (DS)
  - Dedicated iTask server for users in a specific domain (e.g. cs.ru.nl)
  - Own administrations: Login (e.g. rinus@cs.ru.nl), Tasks, Task Instances
  - Static Network: DS's know each others ip-address (global administration)
  - Task (closure) assigned to a user are pushed for evaluation to the DS of that user
    - Code is assumed to be present, generated from same source

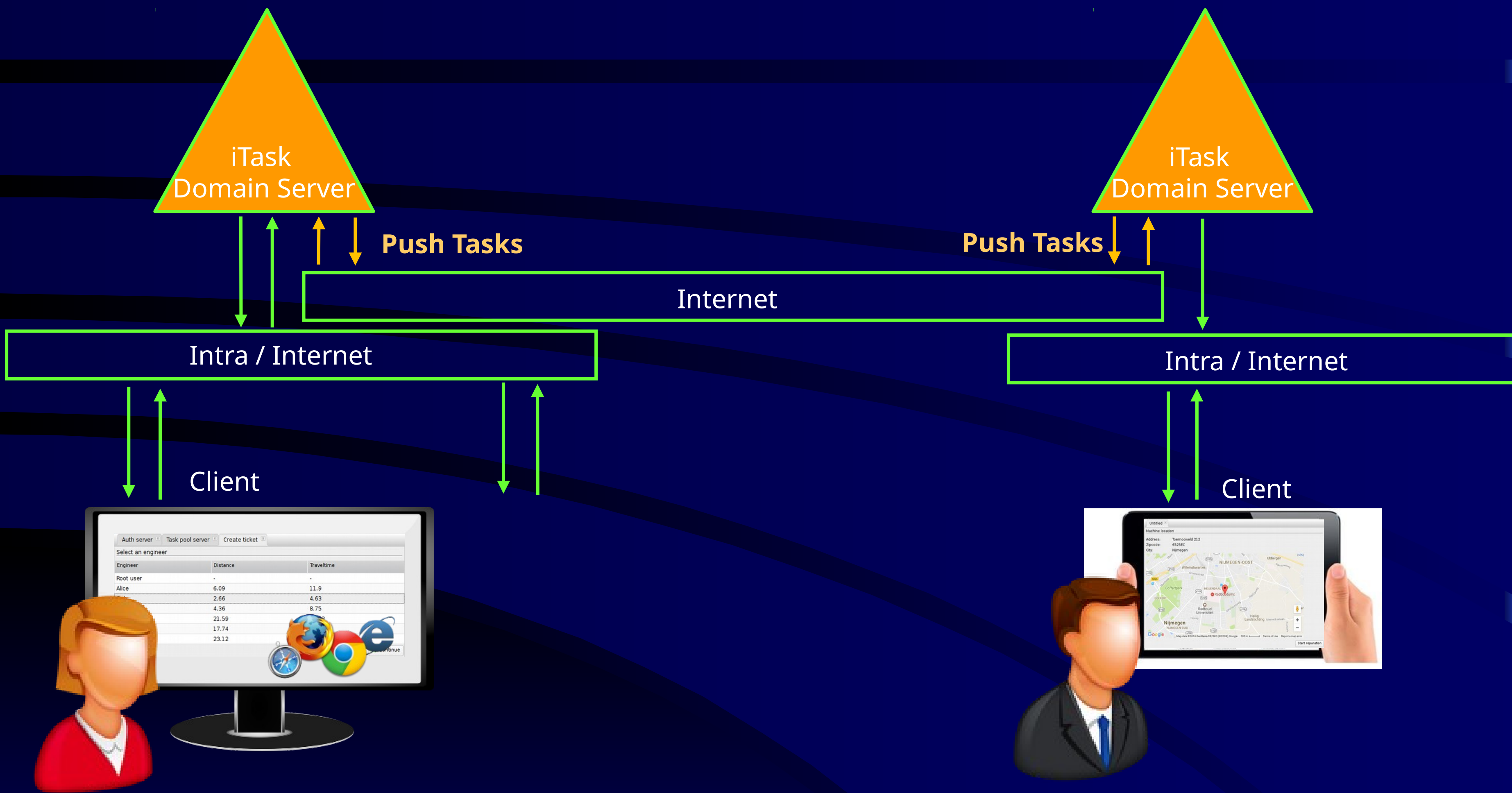
Multiple Domain Servers



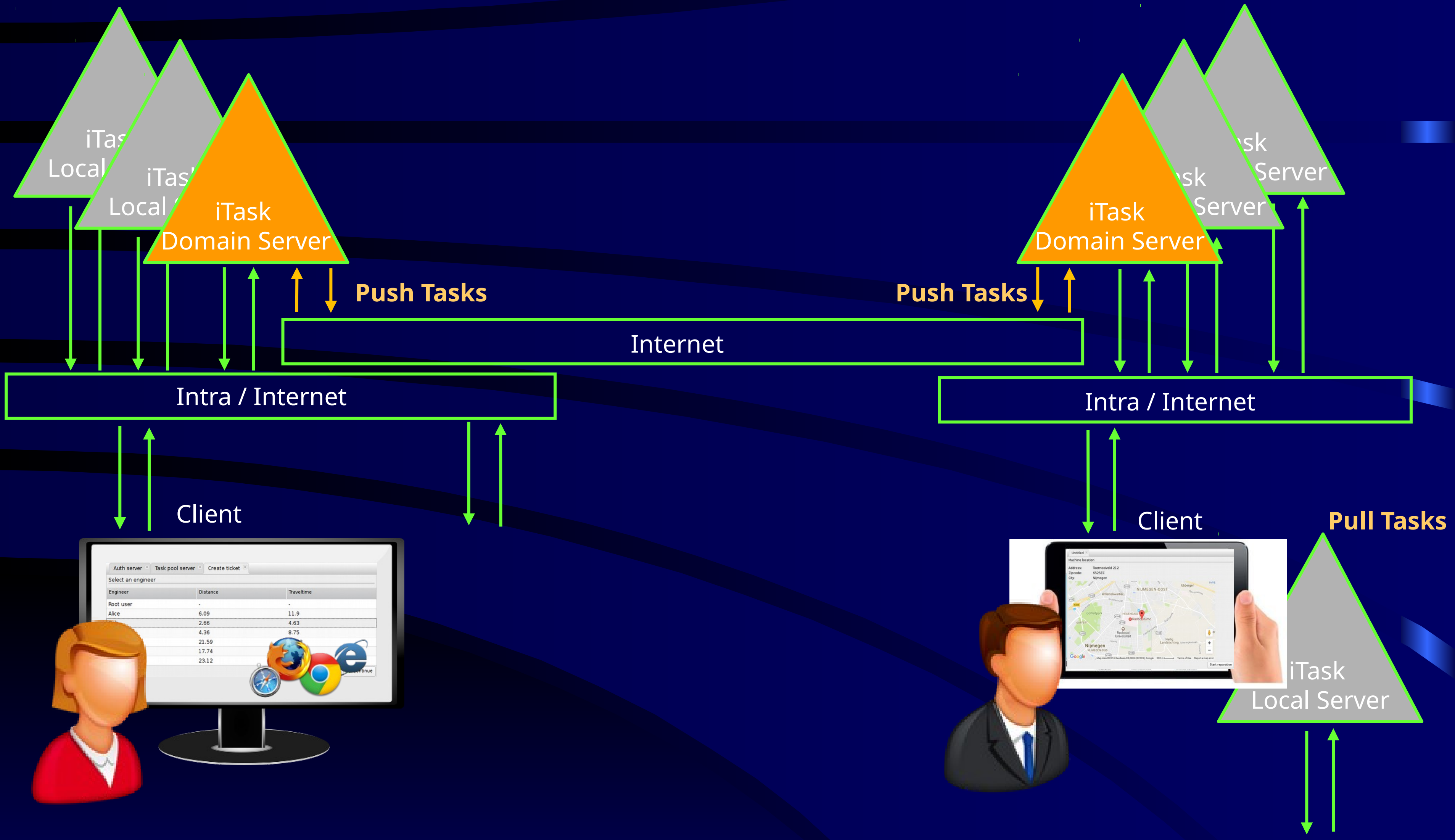
iTask Local Servers

- Delegate tasks of DS to Local Servers (LS)
  - Dynamic Network
  - LS logs-in to its DS (directly or via another LS)
  - An LS can run on a Server (load balancing) or on a Client (allows off-line working)
  - An LS can subscribe to specific tasks of its user, by sending a predicate to its DS
  - All (future) tasks satisfying the predicate are downloaded to the LS

Domain Servers



Domain Servers + Local Servers



Advantage of the Distributed Architecture

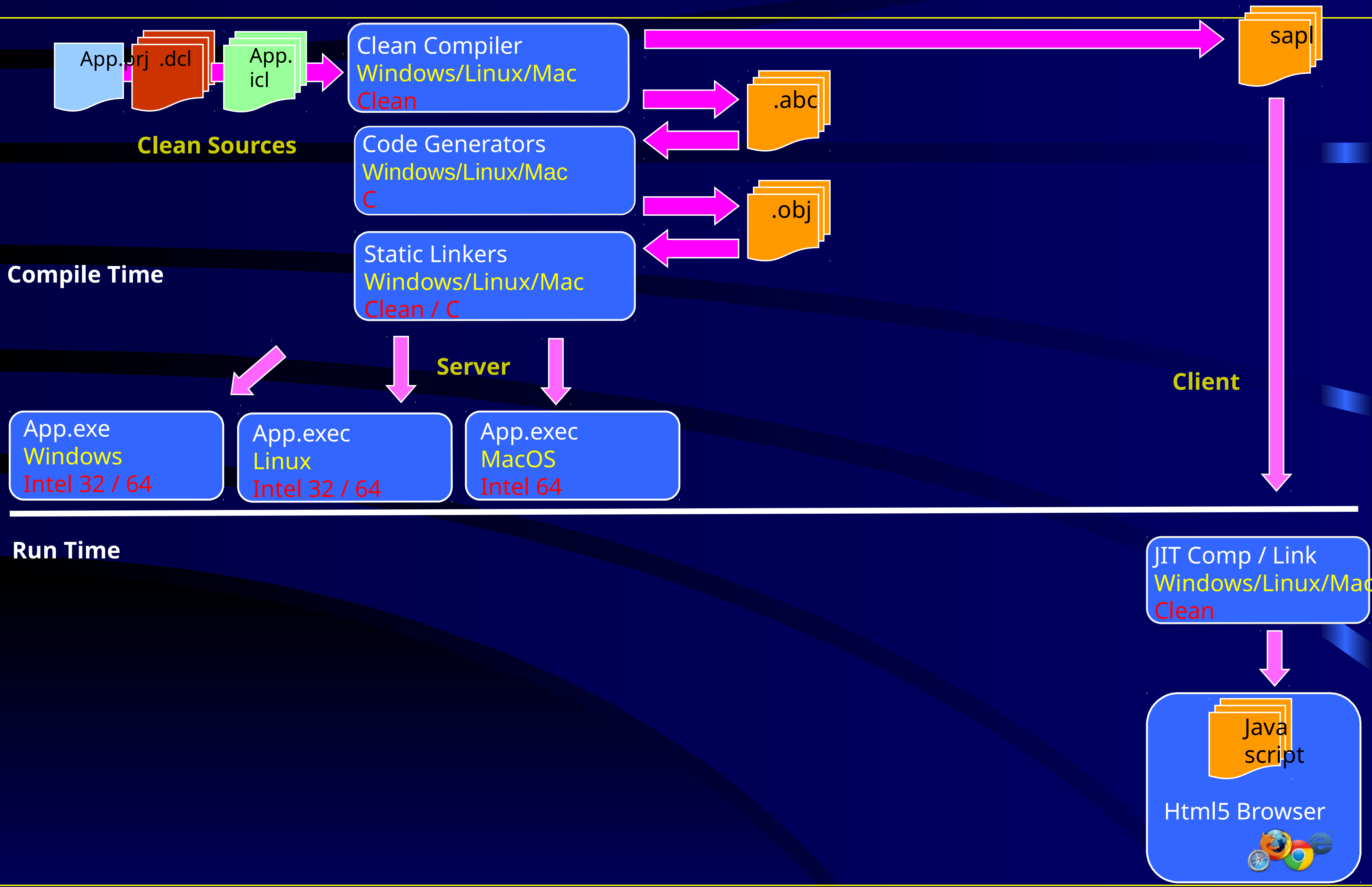
Distributed Architecture with Domain Servers & Local Servers solves all mentioned issues

+ It is a real extension	with 1 DS
+ Runs on all major architectures	Intel / ARM / ARM Thumb
+ Practical advantages	DS for handling users in a domain
+ Scalable	add LS's to a DS
+ One can work off-line	LS on client
+ It's faster	Native code, LS on client
+ Access to all resources	Native code, LS on client
+ One can create self contained "apps"	Native code, LS on client, WebView

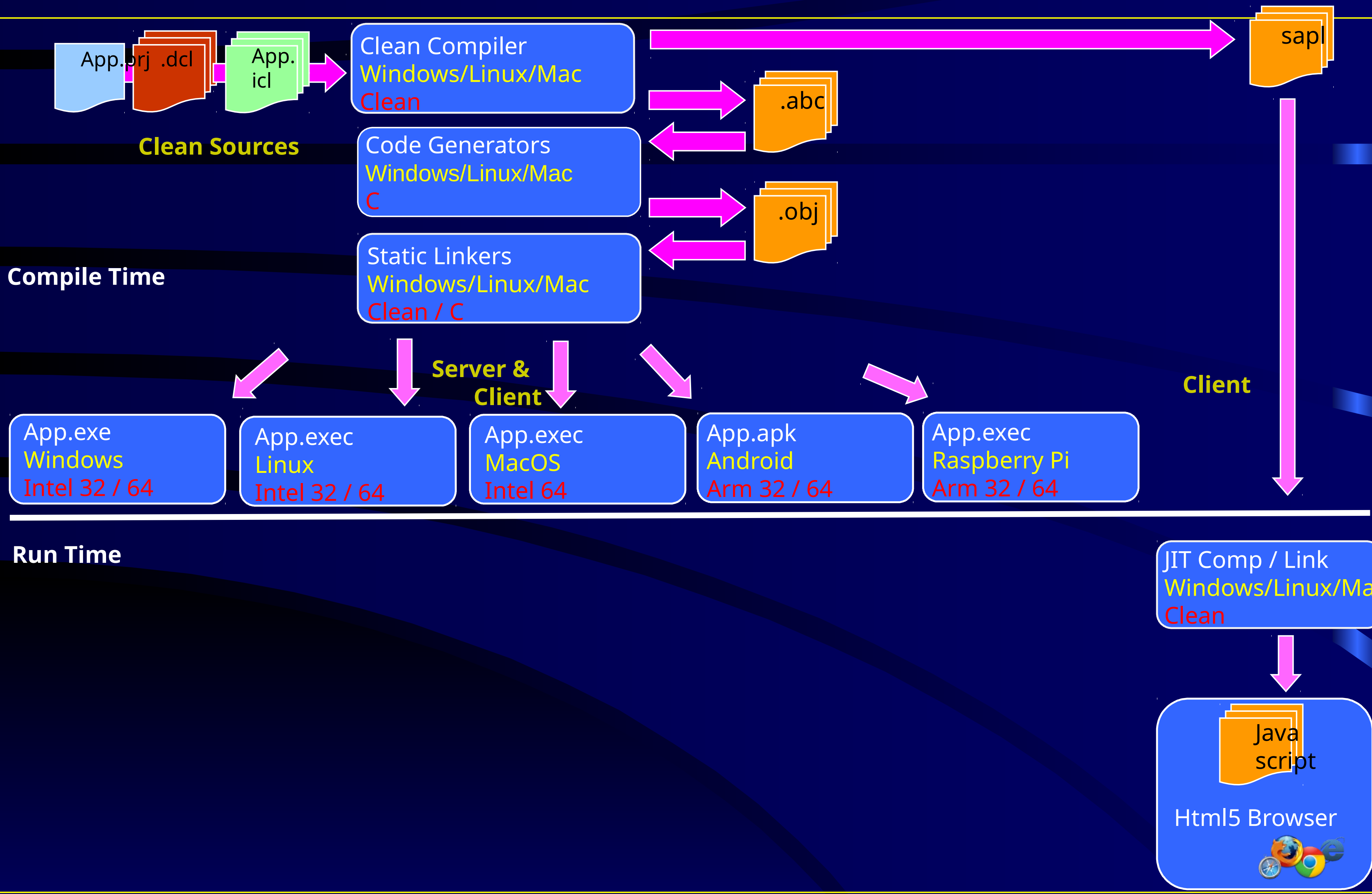
Implementation Challenges (1)

1. Additional code generators needed or ARM – ARM Thumb, Thumb2 – 32 & 64 bit versions

1 Source Solution – Compiled twice for Server & Clients



1 Source Solution – Compiled n-times for Server & Clients



Implementation Challenges (2)

2. Ability to send over *any* functions (closures) for remote evaluation at *any* time from / to *any* platform

- Closures can be *dynamically* constructed, *not* a simple Remote Procedure Call
- No interpreter / one virtual machine, but *native* code
  - Each platform differs in code / stack lay-out / heap lay-out / calling conventions
- All platforms: encoding / decoding of closures to a platform independent symbolic format (graph)

Remote Servers can be asked to handle complex requests by sending over closures

- *Tasks subscription, load balancing, task value synchronization, SDS synchronization*
- *Allows elegant implementation*

Tasks Related Implementation Challenges (3)

*

3. Ability to type-safely send over *Tasks* for remote evaluation at *any* time from / to *any* platform

:: RemoteTask = E.a : RTask (Task a) TaskName TaskId & iTask a

- Cleans ADT to the rescue: adds required dictionaries automatically

4. One has to synchronize Tasks Values produced by Tasks send over for remote evaluation

- Observed on the Sending Server
- Sending Server maintains a copy in an SDS, updated only when the Task Value changes

5. One has to synchronize SDS-values over *all* remote servers having Tasks depending on them

- Distributed versions of set, get, update, watch
- All involved servers maintain a copy of the remote SDS in an SDS
- Lean and mean notification to reduce network traffic
- Asynchronous SDS communication needed, getting complex for parametric lenses (Haye Bohm)

Overview

- Introduction to Task Oriented Programming
- iTask Overview
  - Task Values
  - Editors
  - Task Combinators
    - Sequential Combinators
    - Parallel Combinators
  - Shared Data
  - Current Research

What is the Semantics of an iTasks application ?

- What it the precise meaning of if I define an iTasks application?
- Can I reason about an iTasks application ?
  - is the expression (s -||- t) equivalent to (t -||- s) ?
  - is the expression (s -&&- t) equivalent to (t -&&- s) @ (\(t,s) → (s,t)) ?
  - what is the value of (return 0 -||- return 42) ?
  - is it possible to edit a value if one of the editors is done in

    (task2 = updateInformation "e1" [] 1) -&&- (updateInformation "e2" [] "a“)
- What is the final result of a specific iTask application ?
- Will a certain task be executed in all possible scenario’s ?
- Will it produce the task value I am expecting ?

Editors and task combinators do many things …

- iTask has to handle complex situations

  - synchronize state in editors / tasks with their view in the browsers
  - produce proper javascript code + html-code for the browser
  - handle inputs from the web, update task administration on server
  - generate efficient updates of the corresponding views in the web browser
  - interface and synchronize with files and databases
  - handle input & output of multiple users and systems
  - handle client/server communication & synchronisation
  - handle distributed node servers / clients
  - handle failure and recovery of the devices / clients / servers
  - ..

- iTask is a actually a distributed, platform independent, Operating System

  controlling processes with complicated interactions

Swiss Army Knife Core Combinators

One editor:

```
interact :: d EditMode (RWShared () r w)
        (r → (l, v))                          //On init
        (v l v → (l, v, Maybe (r -> w)))       //On edit
        (r l v → (l, v, Maybe (r -> w)))       //On refresh
        (Maybe (Editor v))      → Task (l,v)
                  | toPrompt d & iTask l & iTask r & iTask v
```

One Sequential Combinator:

```
step :: (Task a) ((Maybe a) → (Maybe b))
        [TaskCont a (Task b)]
                  → Task b         | TC a & JSONDecode{|*|} a & JSONEncode{|*|} a
```

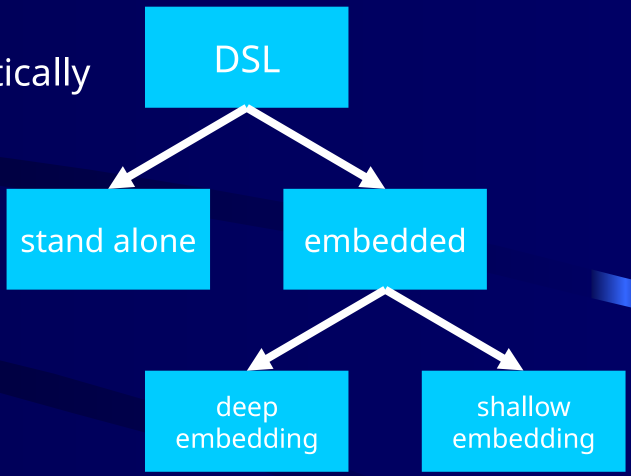
One Parallel Combinator:

```
parallel :: [(ParallelTaskType,ParallelTask a)]
        [TaskCont [(TaskTime,TaskValue a)] (ParallelTaskType, ParallelTask a)]
                  → Task [(TaskTime,TaskValue a)]
                                  | iTask a
```

Semantics

The iTask system is quite a complicated system

- Tasks are reactive and observable
- Shared Data, automatic Publish / Subscribe system
- Multi-user, distributed, multi platform, client-server architecture
- All communication / Storage / Persistence / GUI's handled automatically



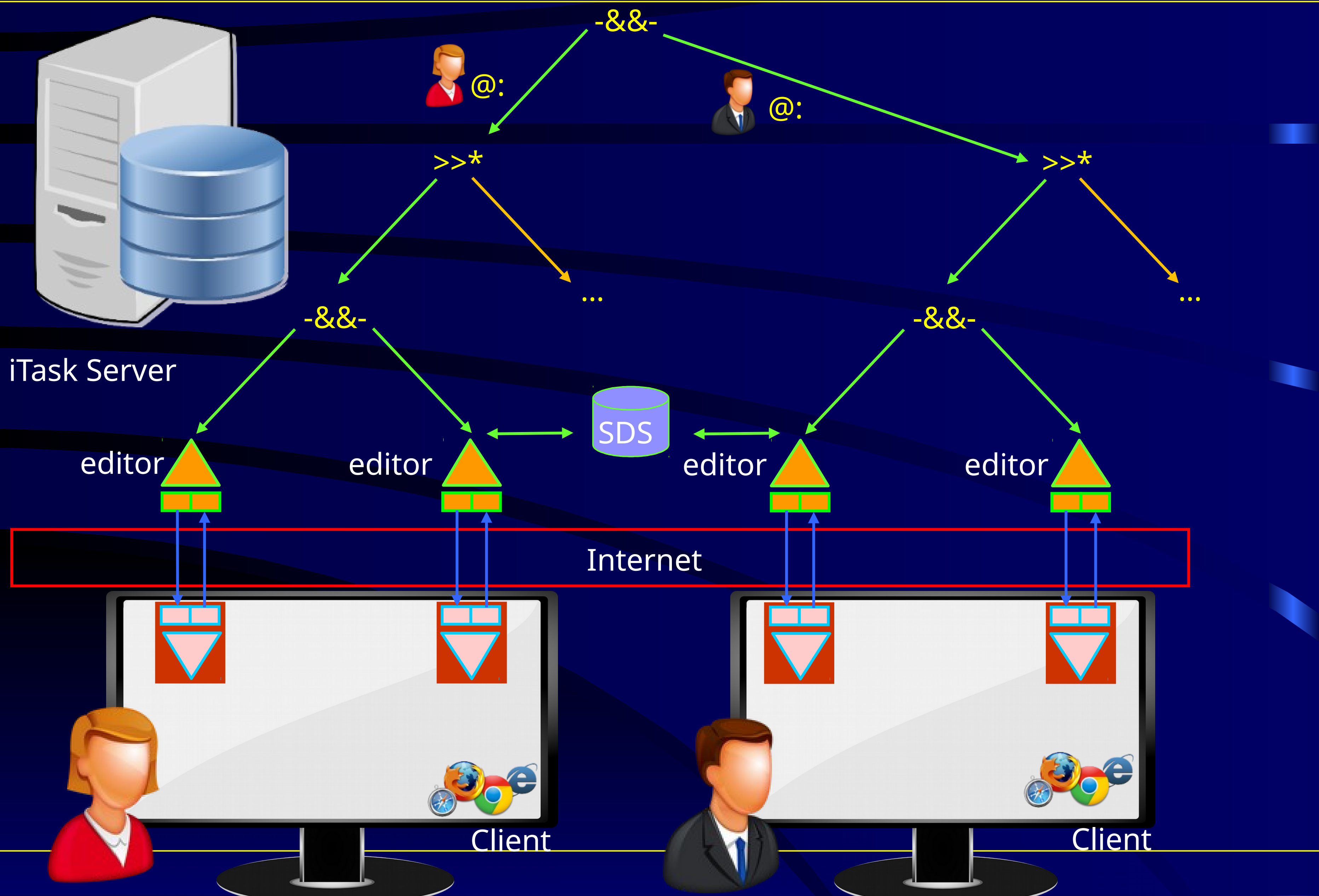
Implementation

- iTasks is a shalllowly Embedded Domain Specific Language
  - Embedded: both DSL and host language,
    - limited syntax options for DSL
  - Shallow embedding realized by using functions
    - other interpretations (e.g. for analysis) difficult
  - Alternative: deep embedding: Algebraic Data Types + interpreter(s)
    - Type restrictions, e.g. all elements of a Tree a has to be of the same type.

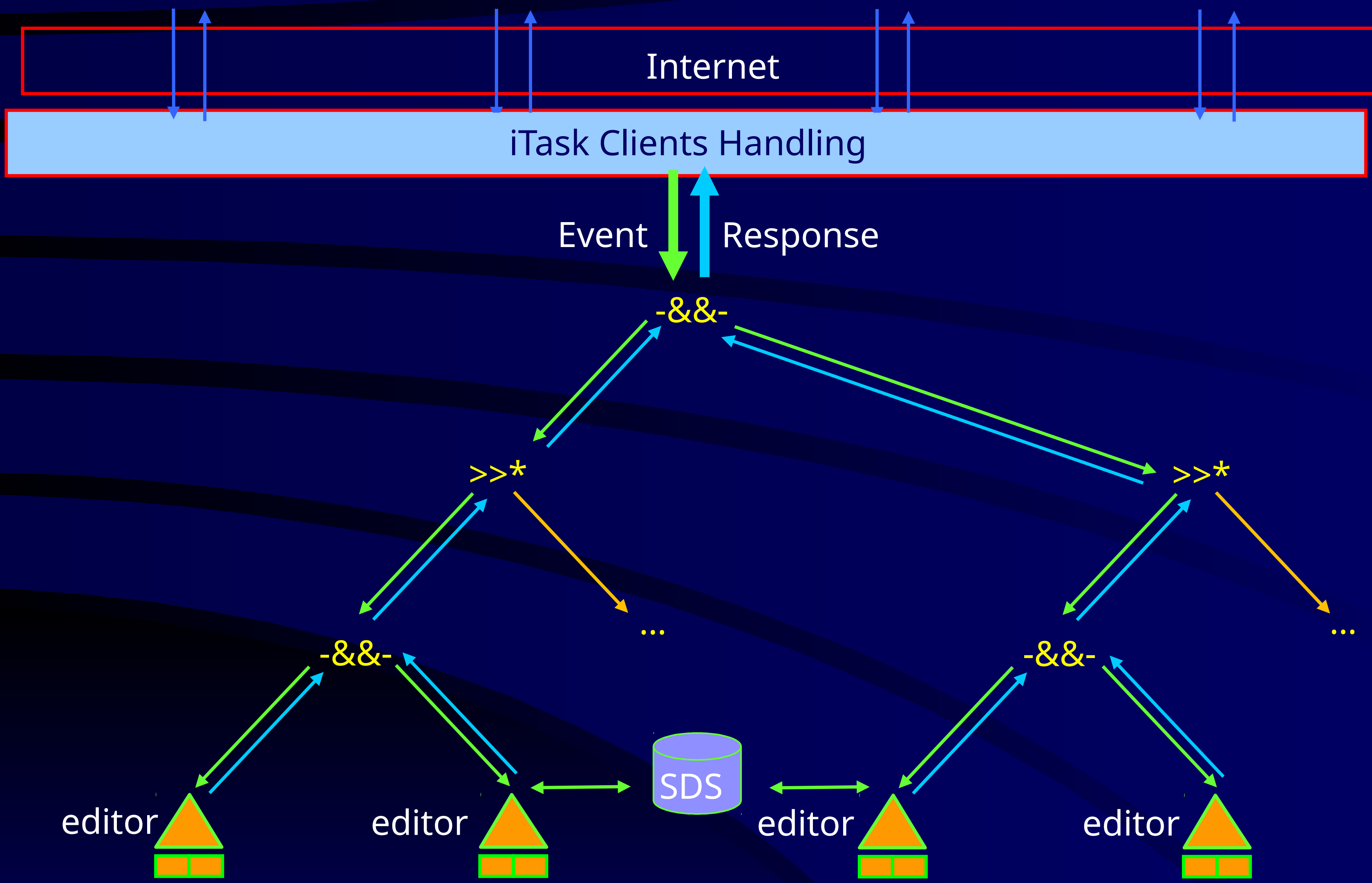
We need Semantics of a Simplified Version of iTasks..

- Operational Semantics described in Clean
- Readable, type checked, concise (a couple of pages of code)
- Executable, one can check and (model based) test its behavior
- Blueprint for actual implementation

iTask Architecture



iTask Architecture



Semantics - Events

```
:: Event     =   RefreshEvent
             |   EditEvent        TaskNo  Dynamic
             |   ActionEvent      TaskNo  Action

:: Action    =   Action String
```

Events can be

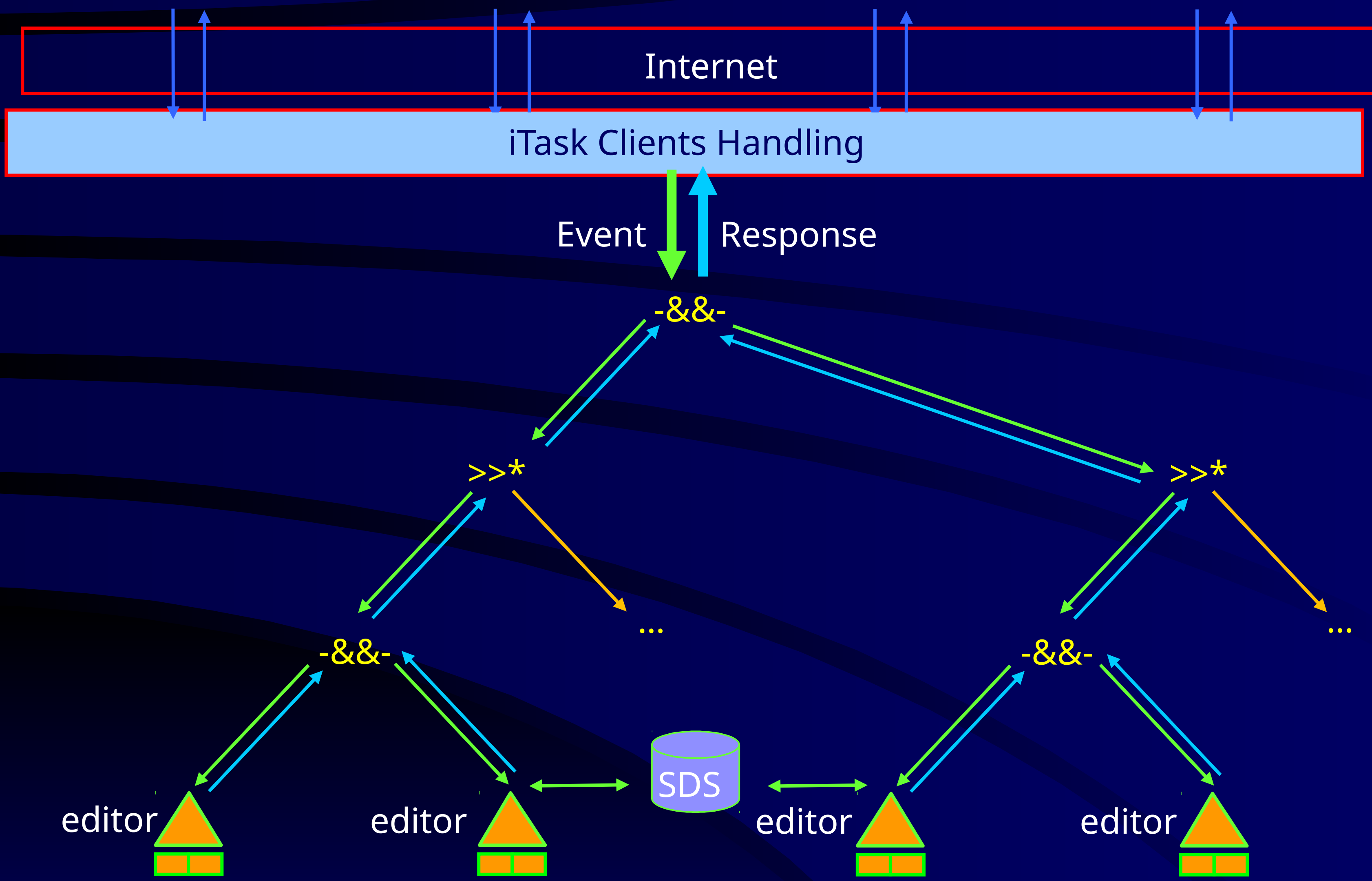
- A general RefreshEvent, used to calculate a new page from scratch
- An EditEvent i.e. some task value changed by some user using some editor
  - the task number TaskNo identifies the task for which the event is intended
  - Task values are stored in a Dynamic,
    such that can be stored in any data structure without causing type problems
- Some ActionEvent trigered by some user
  - the task number TaskNo identifies the task for which the event is intended

Each event is passed around the task tree in preorder on search for the corresponding task for which the event is intended

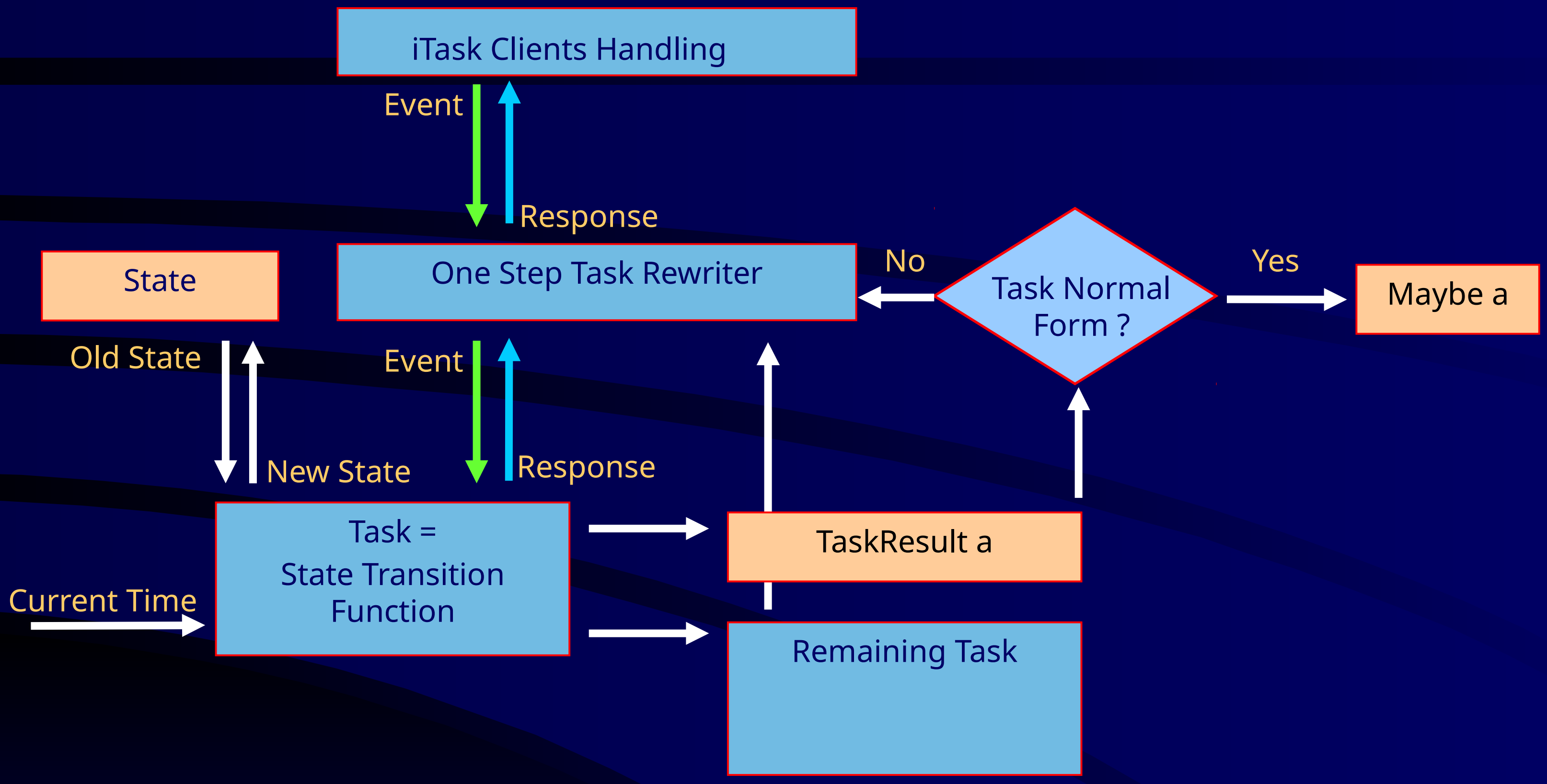
Semantics - Response

```
:: Response          =     EditorResponse        EditorResponse
                     |     ActionResponse        ActionResponse

:: EditorResponse    =     { description      :: String
                     , editValue        :: (LocalVal, SharedVal)
                     , editing     :: EditMode
                     }
:: LocalVal          :==   Dynamic
:: SharedVal    :==  Dynamic
:: EditMode          =     Editing
                     |     Displaying

:: ActionResponse    :==   [(Action, Bool)]
```

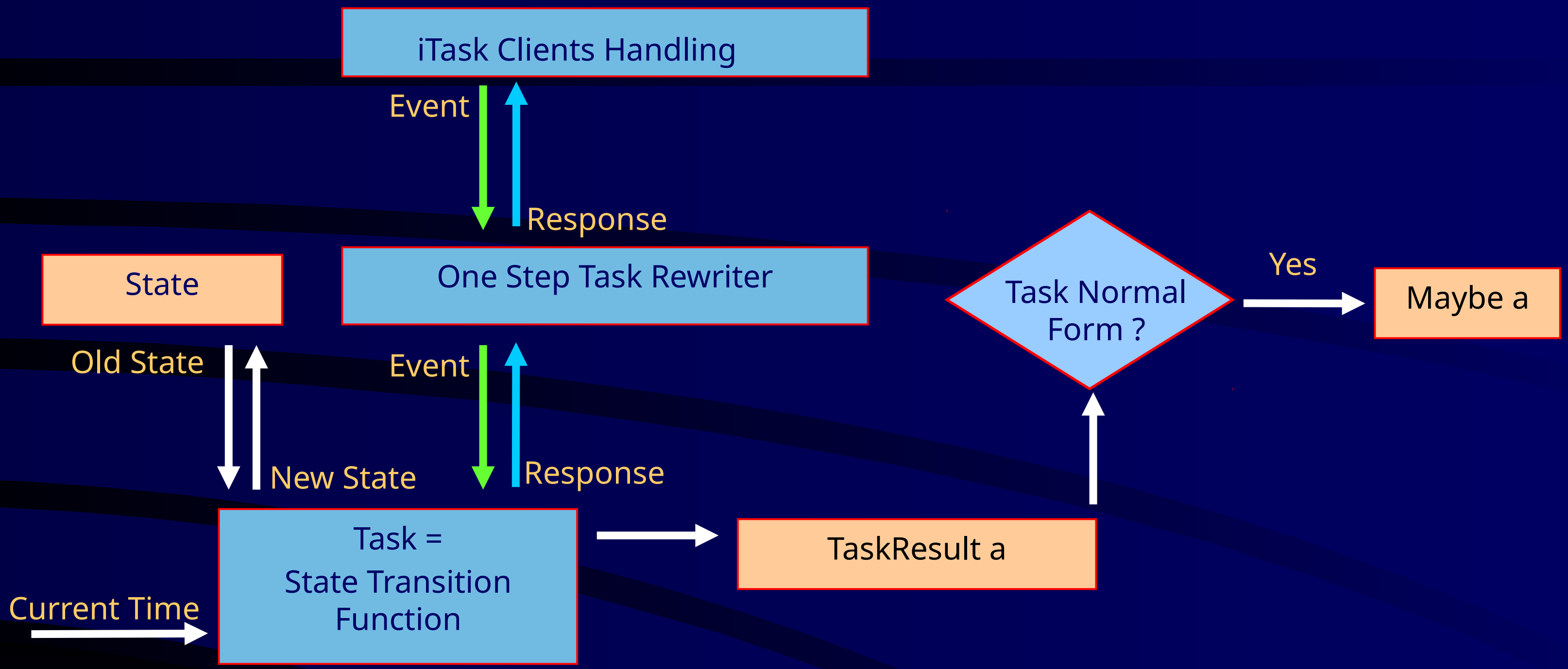
iTask Architecture



Simplified iTasks Architecture



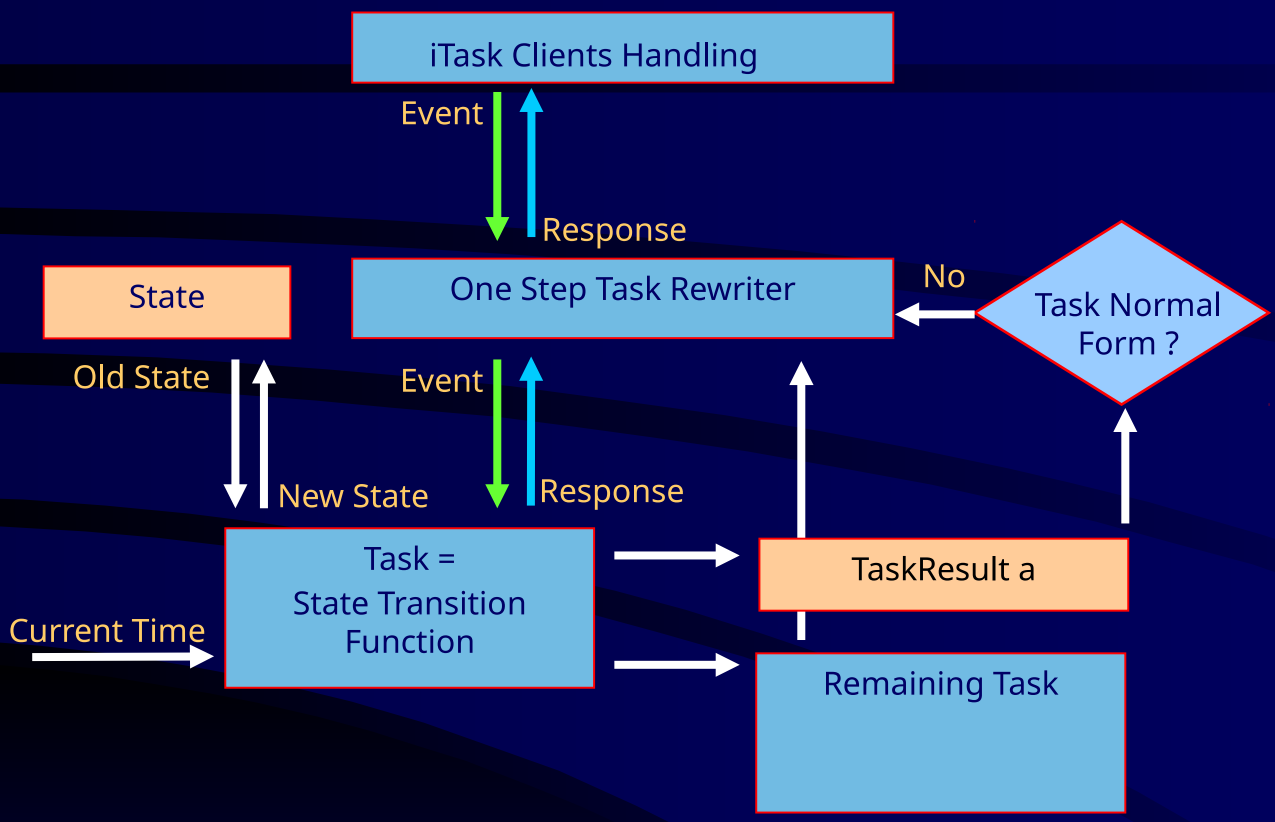
Simplified iTasks Architecture



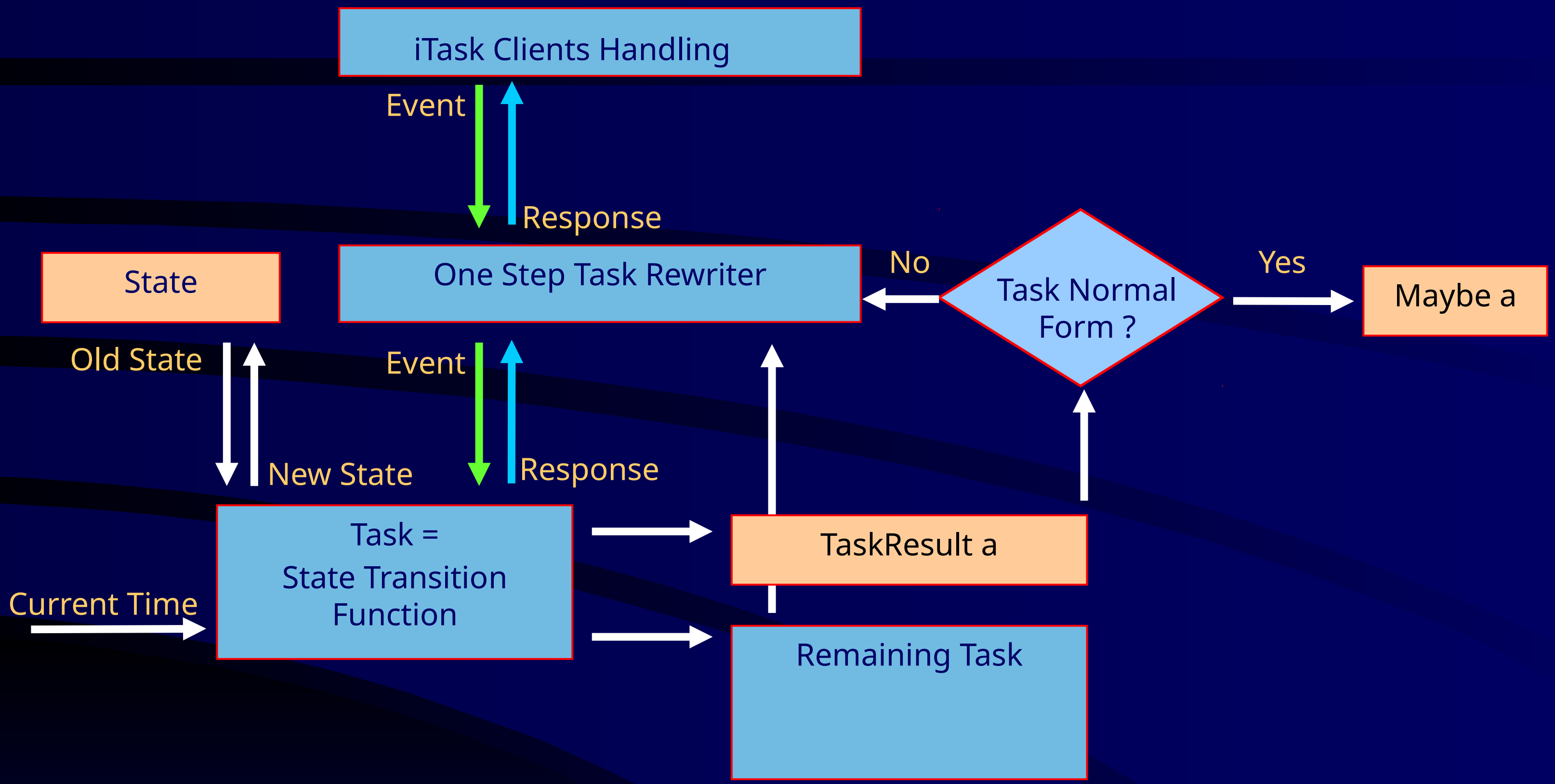
Simplified iTasks Architecture

Maybe a

Simplified iTasks Architecture



Simplified iTasks Architecture

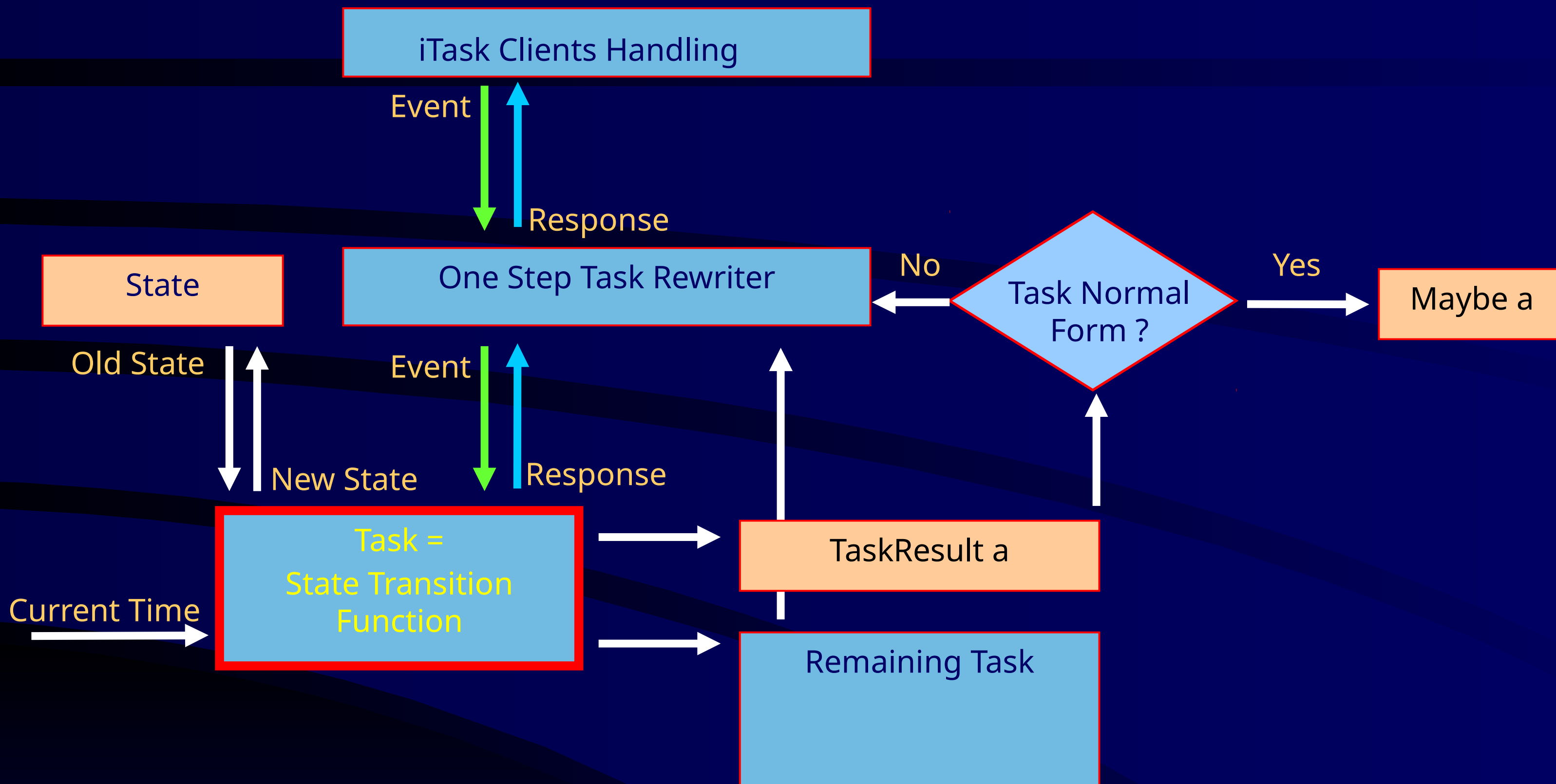


Semantics – State

- Client and Server need to know which task is meant: taskNo (Int) unique identification
- Time can play an important role in tasks: we need to time (TimeStamp) work
- We need a storage to store an arbitrary number of “shares”.
  Shares can be of arbitrary type, static typing of such a storage impossible.
  We need either to fall back to Clean Dynamic types (works for values of *any* type)
  or serialize to JSON code (works only for *first order* types)…
- We need to communicate with the client, save information to disk for persistence, enable I/O…
  We need access to the unique(*) World !

```
:: *State        =    { taskNo     :: TaskNo            // highest unassigned task id
                      , timeStamp  :: TimeStamp         // current time stamp
                      , mem        :: [SharedValue]     // type safe storage for “shares”
                      , world      :: *World            // enables I/O in a pure FPL
                      }
:: SharedValue   :==  Dynamic
```

Simplified iTasks Architecture



Semantics - What is a Task ?

:: Task a   *typed* unit of work which should deliver a value of type a

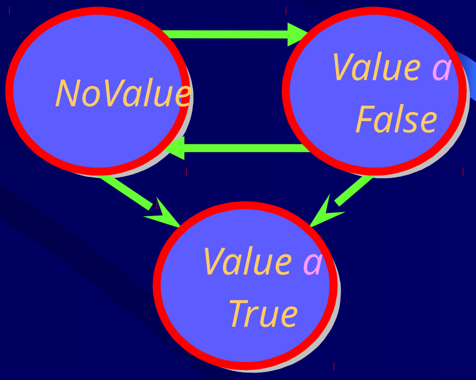
State Transition Function:
i.e. a Monad

```
:: Task a         :==    Event *State → *((Reduct a, Responses),*State)

:: Reduct a              =    Reduct (TaskResult a) (Task a)
```

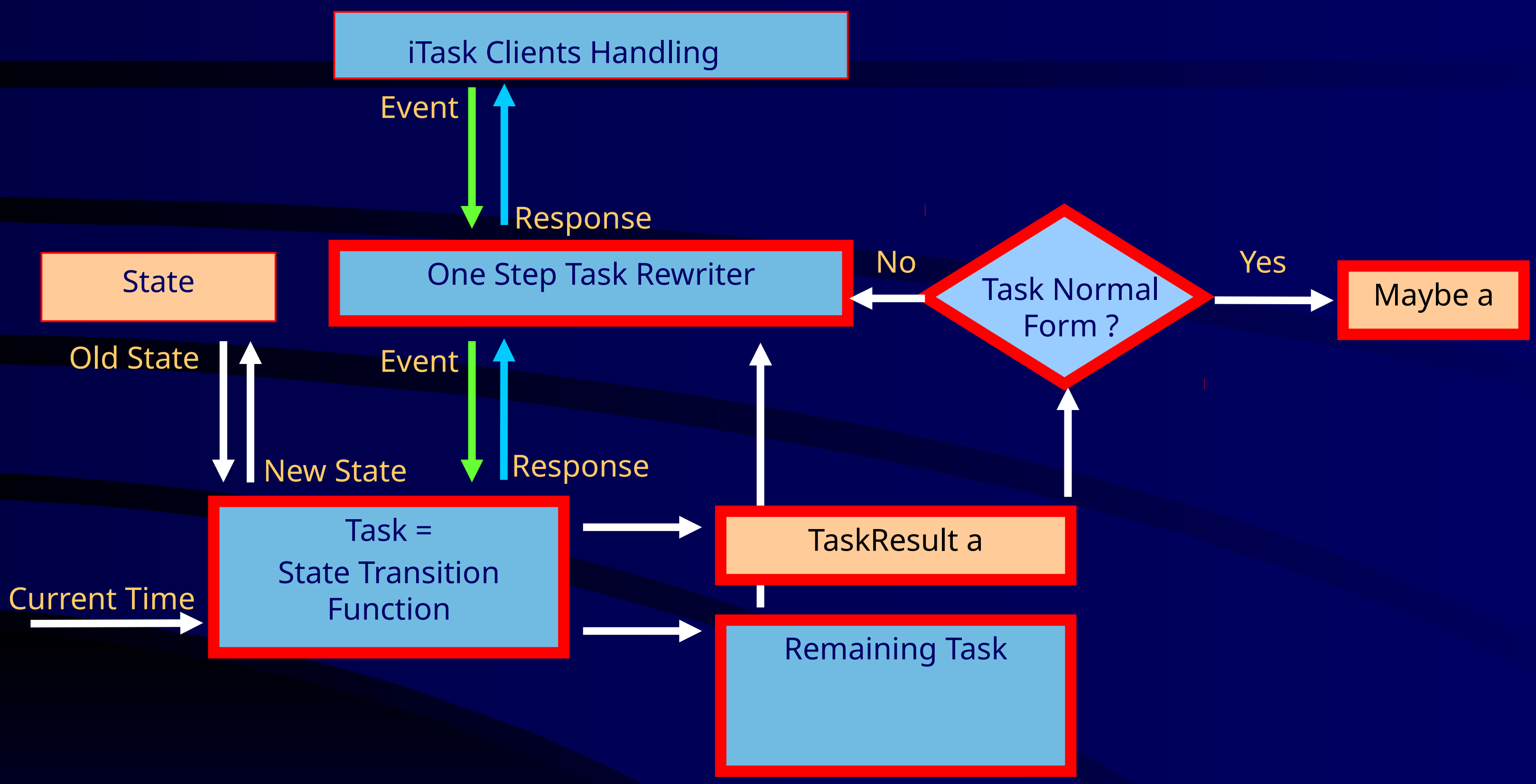
Latest value of a task

Continuation:
remaining task to do

```
:: TaskResult a   =             ValRes TimeStamp (Value a)
                  |    ∃ e:     ExcRes e                     &  iTask e
:: Value a    =   NoValue
                  |      Value  a Stability
:: Stability  :== Bool
```



Simplified iTasks Architecture



Semantics -Task Evaluation by Rewriting

```
evaluateTask :: (Task a) *World → *(Maybe a, *World)          | iTask a
evaluateTask task world
# st          = {taskNo = 0, timeStamp = 0, mem = [], world = world}
# (ma, st)        = rewrite task st                 // evaluate task till normal form is rea
= (ma, st.world )
```

Context restrictions also in semantics used, to enable serialization / de-serialization / equality test

```
rewrite :: (Task a) *State → *(Maybe a, *State)                    | iTask a
rewrite task st
# (ev, nworld)          = getNextEvent     st.world   // wait for next event
# (t, nworld)           = getCurrentTime  nworld   // read the time

                                               // evaluate the task:
# ((Reduct result ntask, responses), st)       = task ev {st & timeStamp = t, world = nworld}

= case result of
    ValRes _ (Val a True)    → (Just a, st)            // normal form reached: stable value
    ExcRes _                 → (Nothing, st)           // oops: un-catched exception has been raised
                                               // not finished: send responses to clients
                                                     // and continue with remaining task
    _                     → rewrite ntask  {st & world = informClients responses st.world}
```

Semantics – Context Restrictions

```
class iTask a | TC a & gEq{|*|} a

serialize :: a → Dynamic                         | TC a
serialize v              = dynamic v

de_serialize :: Dynamic → a                      | TC a
de_serialize (v::a^)     = v
de_serialize _           = abort “Run-time type error”
```

Semantics – Non-Interactive tasks

```
return :: a → Task a
return va = \ev st=:{timeStamp = t} → stable t va ev st
where
      stable t va _ st = ((Reduct (ValRes t (Value va True)) (stable t va), []),st)
```

```
throw :: e → Task a                                                  | iTask a
throw e = \ev st → ((Reduct (ExcRes e) (throw e), []), st)
```

```
(@?) infixl 1 :: (Task a)  ((Value a) → Value b) → Task b            | iTask a & iTask b
(@?) task conv = \ev st
→ case task ev st of
     (Reduct (ValRes t aval) ntask, rsp, nst)
                                   → case conv aval of
                                          Value b True      → return b ev nst
                                     bval          → ((Reduct (ValRes t bval) (ntask @? conv), rsp), nst)
     (Reduct (ExcRes e) _,_,nst)               → throw e ev nst
```

Semantics – Interactive Editors

```
edit :: String l  (RWShared r w) (l  r → Maybe a) → Task a | iTask l & iTask r
edit descr lv share calcValue = newTask (edit1 lv)
where
 edit1 lv myId time ev st
 # (newTime, nlv)
               = case ev of
                   EditEvent taskId dyn → if (taskId == myId )              // edit event for this editor
                                  (st.timeStamp, de_serialize dyn)    // decode new value
                                  (time, lv)                          // value unchanged
                   _                → (time, lv)                      // value unchanged
 # (sr, st)  = share.get st                                           // perhaps share is updated
 #  newValue = toValue (calcValue nlv sr)                             // calc new value edit task
 =  (( Reduct (ValRes newTime newValue ) (edit1 nlv myId newTime)
    , [(myId , EditorResponse  { description = descr
                   , editing       = Editing
                   , editValue     = (serialize nlv, serialize sr)}})]), st )
 where
  toValue :: Maybe a → Value a
  toValue (Just a)  = Val a False
  toValue Nothing  = NoVal


newTask :: (TaskNo TimeStamp → Task a) → Task a
newTask task_fun = \ev st=:{taskNo = no, timeStamp = t}  = task_fun no t ev {st & taskNo = no+1}
```

Semantics – Dealing with Shares

```
:: RWShared r w =   { get ::   *State → *(r,*State)
                    , set :: w *State → *State
                    }

withShared :: a ((Shared a) → Task b) → Task b          | iTask a
withShared va task_fun = withShared`
where
    withShared`ev st
    # (share, st) = createShared va st
    = task_fun share ev st


createShared :: a  *State → *(Shared a, *State)                | iTask a
createShared a st=:{mem}
= ({get = get, set = set}, {st & mem = mem ++ [serialize a]})
where
 idx                = length mem
 get   st=:{mem}    = (de_serialize (mem !! idx),st)
 set a st=:{mem}    = {st & mem = updateAt idx (serialize a) mem}
```

Semantics – Step Combinator I

```
(>>*) infixl 1 :: (Task a) [TaskStep a b] → Task b | iTask a & iTask b

:: TaskStep a b =     OnAction Action (Value a → Bool) (Value a → Task b)
                  |       OnValue         (Value a → Bool) (Value a -> Task b)
                  | E.e:     OnException                    (e → Task b)        & iTask e

:: Action        =     Action String
```

Semantics – Step Combinator II

```
(>>*) infixl 1 :: (Task a) [TaskStep a b] → Task b | iTask a & iTask b
(>>*) task steps = newTask (step1 task)
where
 step1 task myId t event st
 # ((Reduct tval ntask, rsp), st) = task event st
 = hd (findTriggers tval ++ findActions  tval event ++ [step1` tval ntask rsp]) ev st
 where
  findTriggers (ExcRes   e)        = catchers e ++ [throw e]     // find catching step, otherwise propagate
  findTriggers (ValRes _ v)        = triggers v

  findActions (ValRes _ v) (ActionEvent tid act)            // handle actions, if any
  | tid == myId                   = actions act v
  findActions _ _                 = []

  step1` (ValRes _ v) ntask rsp _ st  = ((Reduct no_tval (step1 ntask tn t), nrsp ++ rsp), st)
  where
   no_tval      = ValRes t False
   as           = [(a,p v) \\ OnAction a p _ <- steps]
   nrsp         = if (isEmpty as) [] [(tn, ActionResponse as)]

 catchers e     = [etb e \\ OnException  etb <- steps]
 triggers v     = [atb v \\ OnValue pred atb <- steps | pred v]
 actions act v = [atb v \\ OnAction a pred atb <- steps | act == a && pred v]
```

iTask Semantics Conclusion

The iTasks core consists of only a few concepts…

- editor, shares, combinators (parallel one skipped in slides above)

Operational Semantics iTasks described in Clean:

+ Readable, concise
+ Type checked + Executable
+ It is used as the blueprint for actual implementation: *shallowly* Embedded DSL

Semantic description of iTasks nice example of the expressive power of such a description method

- Monad – State Transition function
- Rewrite Semantics
- Continuation function (remaining things to do)i

The semantic description abstracts from many implementation challenges

e.g. client-server communication, GUI generation, derived combinators, extensions,
combinators for shares, advanced editors (editlets),
efficiency issues, security issues, multi-platform issues, ….

Conclusions

- *Task Oriented Programming*
  - New style of programming for developing multi-user distributed web applications
  - Focusing on tasks, not on the underlying technology
  - All source code in one language

- *Core*
  - reactive tasks working on local and shared data
  - shared data sources abstracting from any type of shared data
  - editor: can handle all interactions
  - sequential and parallel combinators

- *Operational Semantics*
  - defined in Clean
  - readable, concise, type-checked, executable
  - blueprint for implementations