The THINK Component-Based Operating System

Jean-Philippe Fassino - France Telecom, R&D Division

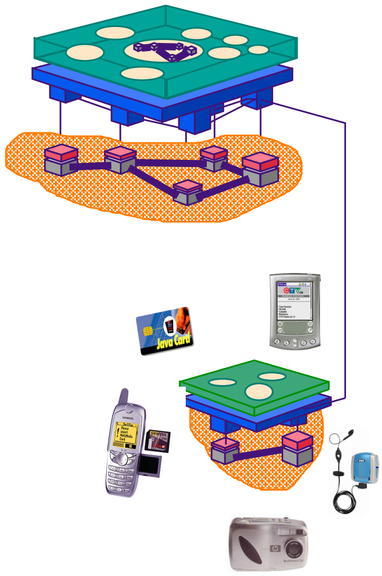
MDE for Embedded System - Brest, September the 6th to 10th 2004
http://www.ensieta.fr/mda/

Partly based on material from T. Coupaye

Overview

Mastering Software Complexity

- **The challenge of software engineering today is to conciliate**
  - the mastering of the
    - development
    - deployment
    - integration
    - management (operation)
    - evolution
  - of software systems that are
    - distributed (asynchrony, faults)
    - heterogeneous (equipments, networks, OS, middleware, services)
    - open (topology, dynamicity)
    - partly legacy (third party elements)
  - … while taking into account applicative services needs (QoS)



- **Where are we ?**
  - *"The computer industry has spent decades creating systems of marvelous and ever-increasing complexity. But today, complexity itself is the problem"* A. Ganek, T. A. Gorbi (IBM)
  - *"We don't understand something very fundamental about how we build systems"* J. Hennessy (Stanford)

Component-Based Software Engineering

- **CBSE is a broad effort towards the *rational construction* and *management* of complex software systems**

- **Preliminary definitions***
  - *"A* **software component** *is a software element that conforms to a component model and can be independently deployed and composed* without modification *according to a composition standard"*
  - *"A* **component model** *defines specific interaction and composition standards"*
  - *"A* **component model implementation** *is the dedicated set of executable software elements required to support the execution of components that conform to the model"*
  - *"A* **component infrastructure** *is a set of interacting software components designed to ensure that a software system or subsystem constructed using those components and interfaces* **will satisfy clearly defined performance specifications"**

**from* G. T. Heineman, W. T. Council (eds.). Component-Based Software Engineering, Putting the Pieces 0. Addison-Wesley, 2001.

Expected Benefits of CBSE (1/2)

- **Adaptation, integration and interoperation of software with bounded effort**
  - Arbitrary deployment environments
    - ↳ construction of dedicated software infrastructures (e.g. embedded operating systems)
  - Evolution in needs, technical evolution
    - ↳ maintainability, durability
  - Organizational evolutions
    - ↳ interoperation, integration
  - Dynamic evolution
    - ↳ availability, scalability

- **Management (administration)**
  - **Uniform control of software resources**: drivers, resources managers, pool, cache, network domain, process, service, …
  - Possibility to **instrument** resources - including dynamically and automatically

Expected Benefits of CBSE (2/2)

- **…more generally, prediction/assessment of certain** *properties* (constraints) **on software** *composition/assembly*
  - Reliability, correctness, fault-management, fault-tolerance
  - Security, access control, isolation
  - Scalability, Availability
  - Testability, manageability, maintainability, durability
  - QoS, real-time
  - …

NB: “Components are a way to impose design constraints that as structural invariants yields some useful properties” Conclusions of the 7th International Symposium on Component-Based Software Engineering (ICSE2004-CBSE7), Edinburgh, Scotland, May 2004

The THINK Initiative

- **THINK Goals**
  - Build complex software by assembling, composing « software bricks » a.k.a. components
  - Provide an *homogeneous vision* of operating systems topology
  - No pre-defined kernel philosophy (e.g mono, micro, exo, …)
  - No pre-defined core functionality (scheduler, memory management, …)
  - Systematic use of components
  - Hardware abstraction components provide direct access to hardware functions
    - Smaller one bring high portability

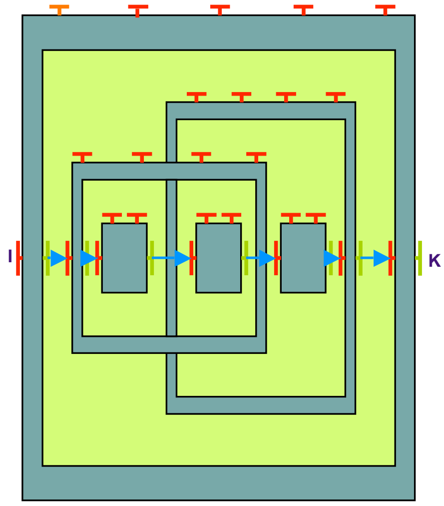
- **Process**
  - Define *architectural principles* and *concepts* for the construction and management of software systems
  - Realize *implementation* than can be
    - Extended
    - Tooled
    - Composed gracefully (communications, security, QoS, …)

- **Adopt the Fractal component model**

Overview

Fractal Concepts Overview

- **« Classical » concepts**
  - ***Components*** are runtime entities
  - ***Interfaces*** are the only access points to components. Interfaces emits and receives operation invocations
  - ***Bindings*** are communication channels. They can be primitive (in the same address space) or composite (to cross over boundaries). In the latter case, they are represented as components and bindings



- **More original concepts**
  - A component is the composition of a ***membrane*** and a ***content***
  - The membrane exercises an arbitrary control over its content. A membrane is made of ***controllers***. It can export control interfaces for some of these controllers
  - The model is **recursive** at arbitrary levels. The recursion stops with base components which have an empty content. Base components encapsulate entities in an underlying programming language
  - Components can be ***shared*** by multiple enclosing components

Components

- **Definition**
  - ***A** runtime **entity exhibiting a recursive structure and reflexive capabilities…***

- **Components as runtime entities**
  - Components are units of
    - Development : design, modelling, implementation
    - Deployment
      - –Packaging, installation, configuration, activation
      - –Operation (administration)
  - Fractal targets the complete software life cycle but focuses on component as runtime entities
      - –**Components do exist and can be manipulated as such at runtime for operation purposes**

- **No predefined « granularity »**
  - Fractal components can be of any size
    - e.g. service, resource, data, protocol stack, packet network, binding (stub), drivers, threads, scheduler, …

Interfaces

- **Definition of (component) interface**
  - ***An access point to a component***
  - *An interaction point between components*
  - *A place where operation invocations car be emitted or received*

- **Not to be confused with *language interface* in relation with typing in Fractal**
  - Defined as a list of *operation signatures*, e.g. IDL interfaces
  - Also called ***interface signature*** in Fractal typing
    - An *interface type* defines
      - –an *interface signature*
      - –some additional *properties (constraints)* (cardinality, contingency…)
    - Which maps ODP definition :

      "An abtraction of the behaviour of an [component] that consists of a subset of the interactions of that [component] together with a set of constraints on when they may occur"

- **Fractal terminology also uses**
  - *Client* versus *server* interfaces a.k.a *role*
    - *A place where operation invocations car be emitted (client interface) or received (server interface)*
  - *"Functional"* versus "*control"* interfaces
  - *External* versus *internal* interfaces (detailed later)

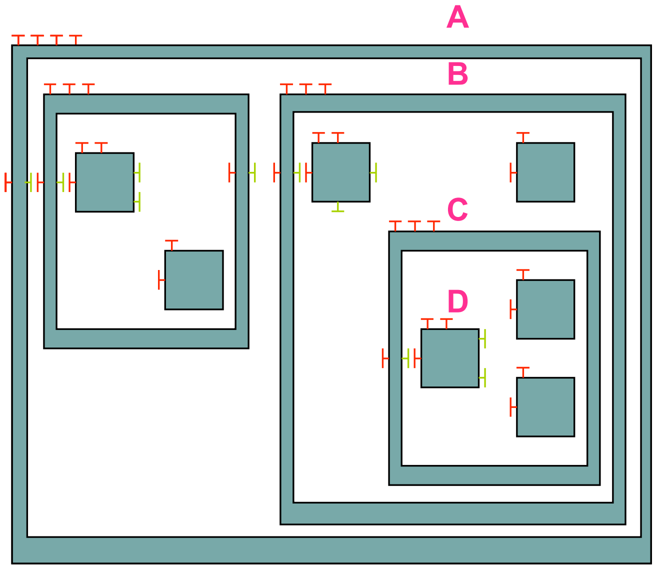
Bindings

- **Abstract definition**
  - ***A communication channel between components***
  - Bindings are "oriented" from a *client* to a server *interface*

- **Bindings are concretized as "*primitive"* or "composite" bindings**
  - (Primitive) binding : local communication channel between components (interfaces), i.e. in the same address space
    - (Primitive) Bindings are typically implemented by C pointers.
  - Composite binding : a configuration of *binding components* and (primitive) bindings
    - Binding components : specialized component dedicated to communication between components to deal with :
      - –Distribution : components not in the same address space
      - –Communication styles : synchronous, asynchronous…
      - –Other communication properties connected to proxy, security, transactions, etc.

Hierarchical Structure

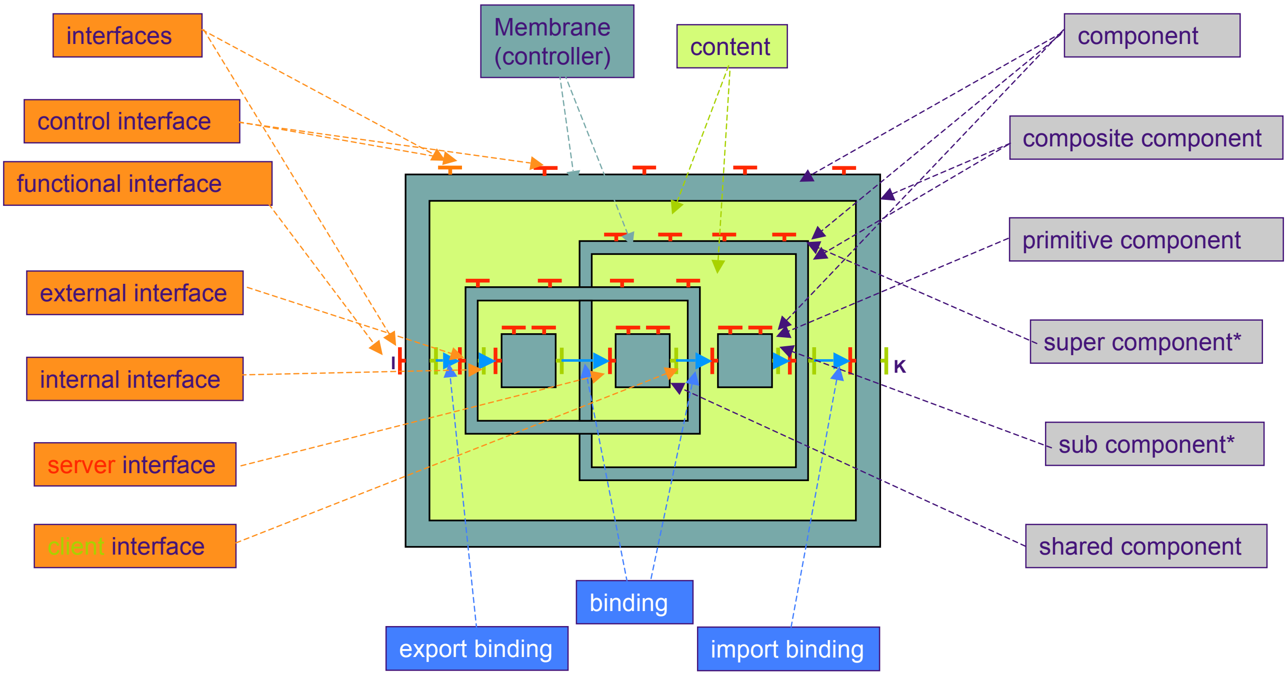
- **Fractal is hierarchical at arbitrary levels**
  - Components as units of configuration
  - A component controls the components it contains (a component is under the control of its enclosing component(s))
- ***Composite* & *primitive* components**
  - A, B, C are composite
  - D is primitive
- ***Sub* and *super* relationship**
  - C is sub component of B
  - C is super component of D



(Reflexive) Control

- **Definition**
  - **A runtime entity exhibiting a recursive structure and reflexive capabilities. *A component is composed of a membrane and a content.*** *A component has well defined access points called interfaces,* ***and provides more or less introspection and control (intercession) capabilities.***
- **A membrane exercises an arbitrary control over its content. It embodies the control behavior associated to a particular component, e.g:**
  - It can provide explicit and causally representation of its content (sub components)
  - It can intercept oncoming and outgoing operation invocations targeting or originating from its content an
    - Superpose a control behavior :
      - –Suspending, check-pointing, resuming activities
      - –Reifying or changing operation invocations parameters
      - –Managing transparently technical services : persistency, security…
      - –Managing QoS : memory consumption (garbage collection)…
  - Or it can do nothing at all!

Structural Concepts and Terminology



Architectural Principles

- **Components as runtime entities**
  - Focus on deployment and **management** of open systems
- **Separation between interfaces and implementations**
  - Interfaces are **the only access points** to components
- **Programmatically controllable bindings**
  - Bindings are not static nor « hidden in code » but externalized so as to be manipulable by (external) programs
- **Hierarchical structure with sharing**
  - Encapsulation, resource management
- **Arbitrary reflexive control**
  - The specification only defines some predefined (“standard”) control interfaces dedicated only to structure management (except perhaps life cycle management …)

Benefits

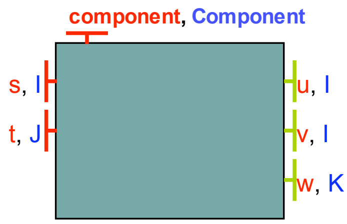
- **For development**
  - The model enforces the use of « good principles »
    - Requires programmers to think about architecture
  - Separation of concerns : functional vs non functional (management)
  - Homogeneous development methods and tools
    - Modelling
    - Test
    - Debugging
  - Reusability, portability
- **For integration, deployment and administration**
  - Reusability (quick portability)
  - Configurability (assemblage, parameterisation)
  - Homogeneous administration tools (supervision, monitoring…)
  - Reconfigurability, maintainability
    - Safety
    - Availability
    - Scalability
    - QoS
- **Altogether**
  - More productivity for programmers
  - More durability for software

Organization of the model

- **Elements of the model specification**
  - « Levels of control »
    - Foundations
      - –Base components with no reflexive capabilities (legacy code)
      - –IDL : Fractal is not only for Java
      - –Naming and binding API (Name, NamingContext, Binder)
    - Introspection
      - –Component and Interface API
      - –Introspection of components boundaries
    - Configuration
      - –(structural introspection and intercession) Attribute, Content, Binding, Lifecycle control API
      - –(Predefined but more generally arbitrary) reflexive control of (white-box) components structure
  - Basic Typing : role, contingency (optional, mandatory), cardinality (singleton, collection)
  - Instantiation
    - Generic factories : create components of a type given as input
    - Standard factories : “ad-hoc” factories that create components of one type each
    - Templates : “facilities” that create components isomorphic to themselves
- **Everything is optional and extensible (« open model »)**
  - Introspection, control interfaces and controllers, factory interfaces, typing

Introspection API

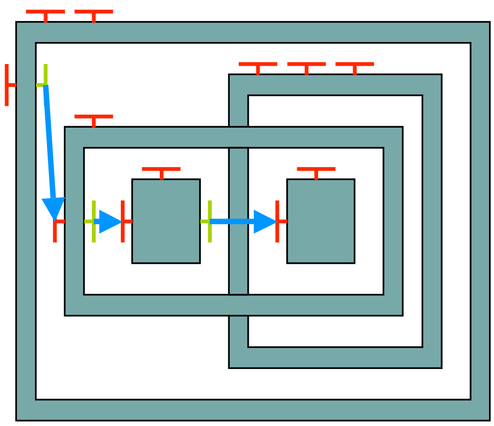
- **This level provides external introspection capabilities**
- **A component at this level owns the `Component` Fractal control interface for interfaces discovery**
  - Component allows discovery of all interfaces owned by a component (server and client, external and internal)
- **Interfaces are named**
  - The name of an interface is valid in the context of the component that owns this interface
- **Interfaces are typed**
  - An interface implements both
    - its functional interface signature (expressed in Fractal IDL: e.g. **I, J, K**)




```
interface Component {
   any[] getInterfaces ();
   any   getInterface (string itfName);
}
```


Configuration API (1/5)

- **This level provides more introspection and intercession capabilities**
- **It allows for exposition and control of components internal structure**
- **It defines 5 « standard controllers » …**
  - `BindingController`
  - `ContentController, SuperController`
  - `AttributeController`
  - `LifeCycleController`



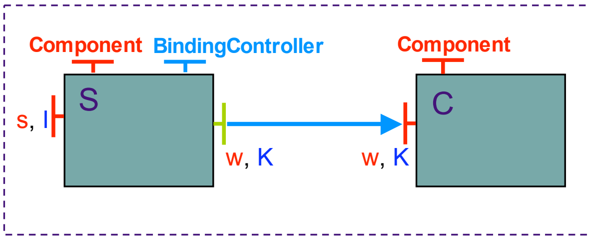
…used for initial configuration and dynamic reconfiguration

- **NB**
  - Bindings and content controls are really central to *architectural configuration*
  - Attributes control is concerned with a restrictive, classical sense of configuration : parametrization
  - Stricly speaking, life cycle control is not concerned with configuration - but it often needed for configuration

Configuration API (2/5)

BindingController

- Used to manage *local bindings* between components
  - Complex bindings (e.g. remote)
    - –Must be created using the naming and binding framework
    - –May involve binding components
  - Strong semantics of *locality*
    - –Bound interfaces must be owned by components in a same direct enclosing component




```
interface BindingController {
  void bind (string c, any s);
  void unbind (string c);
  string[] list();
  any lookup (string c);
}
```

Binds the client interface named c of C to the server interface s of S in the same

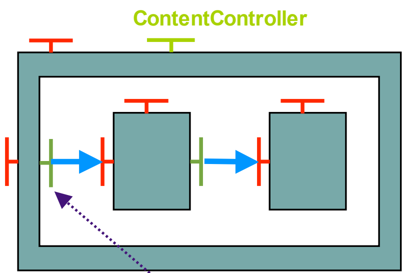
Lists the names of client interfaces of the component

Returns the client interface of a given name

Configuration API (3/5)

ContentController

- Used to manage the hierarchical structure of components
- Management of bindings between sub and super components (a.k.a. import/export bindings) belongs to content control




```
interface ContentController {
  Component[ ] getSubComponents();
  void addSubComponent (Component c);
  void removeSubComponent (Component c);
  any[] getInternalInterfaces();
  any getInternalInterface (string c);
}
```

Lists the sub components of the component's content

Adds and removes a sub component from a component's content

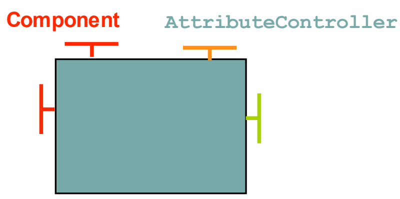
Access to internal interfaces of the component for import/export

Configuration API (4/5)

AttributeController

- Attributes are configurable properties of components
- AttributeController interface can be extended
  - Actually, only integer type managed

```
interface AttributeController {
    int get(string attr);
    void set(string attr, int value);
}
```



Configuration API (5/5)

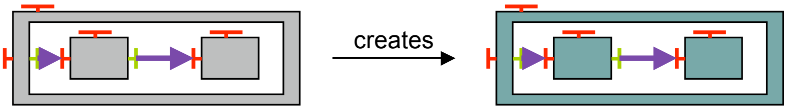
LifeCycleController

- Semantics of start and stop are voluntarily as weak as possible
  - May implement usual suspend/resume or start/stop semantics
  - May or may not start/stop sub components
- The central point is the isolation of 2 states
  - STARTED in which components can accept operations only through their functional interfaces
  - STOPPED in which components can accept operations only through their control interfaces
- LifeCycleController will often be extended or completely redefined to suit arbitrary life cycles
- *NB: this part of the specification may evolve and even disappear…*

```
interface LifeCycleController {
    void start();
    void stop();
}
```

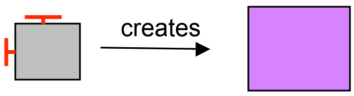
Instantiation API

- **Template**
  - Component description
  - Can be instantiated many times
    - Statically (by compiler)
    - Dynamically (through generated factories)



- **Factory**
  - Dynamically create components of one specific type
    - They are explicitly programmed to do so
    - Or they are generated from template component description

```
interface Factory {
    Component newInstance();
}
```



Overview

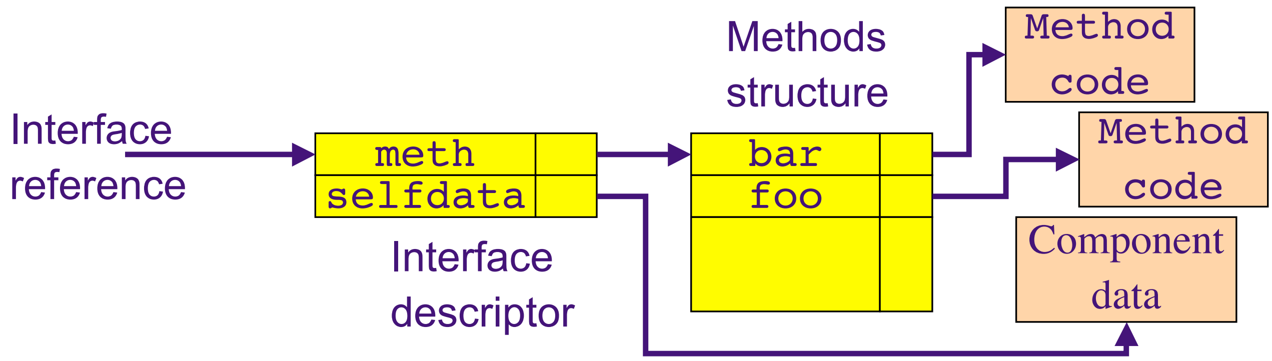
- **I - THINK and CBSE**
  - Mastering software complexity
  - Component-Based Software Engineering
  - Expected benefits
  - The THINK initiative
- **II - Fractal component model**
  - Concepts
  - Principles
  - Organisation (open component model)
  - APIs
- **III - THINK, a Fractal support for Operating System**
  - Interfaces
  - Components
  - Development
  - Configuration
  - Deployment
- **IV - The THINK component library**
  - The library
  - Supported hardware
  - Typical composition
  - Evaluations
  - Kernel constructions
- **V - Conclusion**
  - Summary
  - Links

Interface Definition Language

- **Interface Definition Language (IDL)**
  - Similar to ODP
  - Interfaces are expressed with
    - Constant fields
    - Methods
    - Types
      - Primitives
        - [unsigned] {byte, short, int, long}, char
        - any
        - string
      - Interface references
      - Arrays
  - Programming languages mapping
    - C
      - Primitive xxx ⇒ jxxx, unsigned xxx ⇒ juxxx
      - any ⇒ void*
      - string ⇒ char*
      - [ ] ⇒ *
      - …

```
interface Test {
  int          bar(int arg);
  unsigned int foo();
}
```


Instantiation of interfaces
(cf. C++ Vtable)



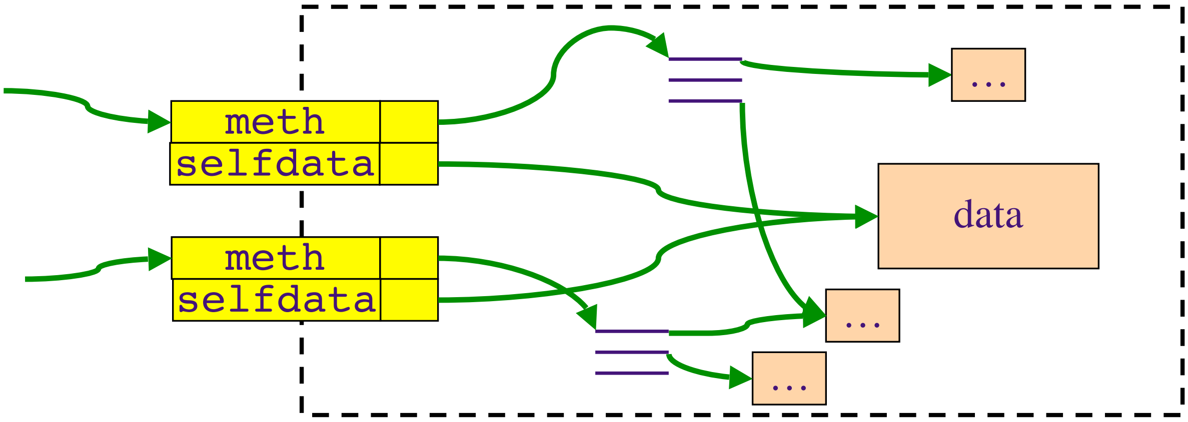

```
interface Test {
  int bar(int arg);
  int foo();
}
```

```
typedef struct {
  struct Mtest* meth;
  void *selfdata;
} RTest;
```

```
struct MTest {
  jint (*bar)(void* this, jint arg1);
  jint (*foo)(void* this);
};
```

Instantiation of components

- A component owns one or more functional interfaces



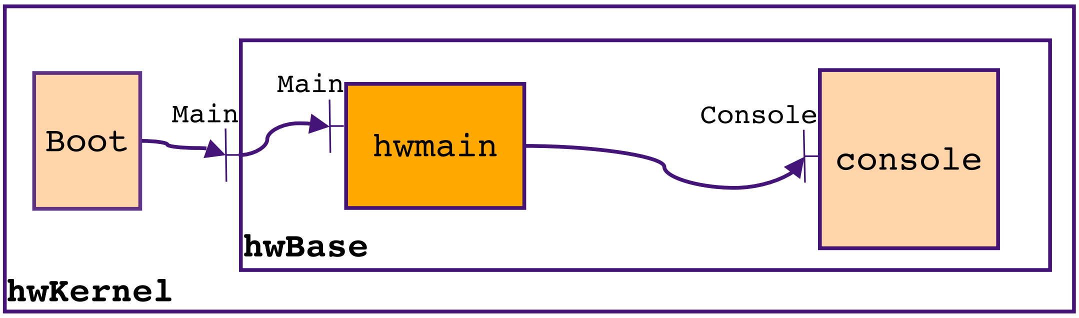
- Interface calling

```
CALL(itf, proc, ...)
```

```
itf->meth->proc(itf->selfdata, ...)
```


Architecture Design

- The `Hello World` example




```
package activity.api;
interface Main {
  void main(int argc,
            string[] argv);
}
```

```
package video.api;
interface Console {
  void putc(char c);
  void putcs(string str);
  void putxycs(int x, int y,
               string str);
  void scrollup();
  int  cols();
  int  rows();
}
```

Primitive ADL : hwmain

```
type MainType {
      provides activity.api.Main as main
}

type hwType extends MainType {
      requires video.api.Console as console
      attributes position
}

primitive hwmain implements hwType {
      skeleton hw nolifecycle
}
```

Can be mandatory, optional or dynamic

or

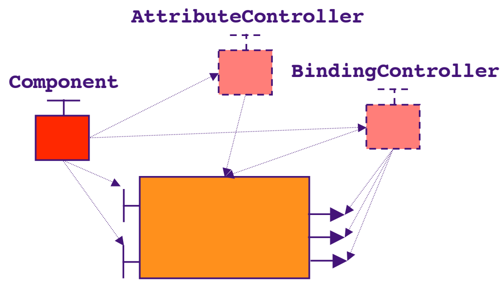
```
primitive hwmain {
      requires video.api.Console as console
      provides activity.api.Main as main
      attributes position
      skeleton hw nolifecycle
}
```


Primitive implementation : hwmain

```
#include <activity/api/Main.idl.h>
#include <video/api/Console.idl.h>

struct hwmaindata {
      // Imported interfaces
      Rvideo_api_Console*      console;
      int    position;
};

#if ! defined(ONLYDEFINITION)

/* Main interface method */
static void mainentry(void* _this, jint argc, char** argv) {
      struct hwmaindata* self = (struct hwmaindata*)_this;
      CALL(self->console, putxycs,
            self->position,
            self->position, "Hello World !");
}

struct Mactivity_api_Main hwmain_mainmeth = {
      main: mainentry
};

#endif
```

Providing Fractal Configuration API

- **Either delegate configuration API implementation**
  - Generic shared tiny code
    - Works on meta-data which encodes internal & external behaviors extracted from ADL
      - –Provided & required interfaces
      - –Attributes
  - Meta-data can be generated by tools
  - Need to respect some programmatic convention naming



- **Or implement our own**

Delegation mechanisms

```
struct exportedskeleton {
  Rfractal_api_Component  myreference;
  unsigned int nbexported;
  struct {
    char*     name;
    struct { void* meth, *selfdata;} itf;
  } exported[];
};
```

```
struct importedskeleton {
  unsigned int nbimported;
  struct {
    char*     name;
    void*     itf;
  } imported[];
};
```

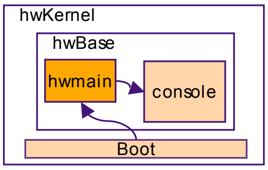
```
/* BindingController delegation implementation */
static void bind(void* _this, char* name, void* itf) {
  struct importedskeleton *self = _this;
  int i;
  for(i = 0; i < self->nbimported; i++) {
    if(strcmp(self->imported[i].name, name) == 0) {
      *(void**)(self->imported[i].itf) = itf;
      return;
    }
  }
  // Error: trying to bind an unexisting interface name
}

struct Mfractal_api_BindingController _bindingcontroller = {
  bind: bind,
  …
};
```

Generated meta-data : hwmain

```
#define ONLYDEFINITION

#include <hwmain.c>
static struct hwmaindata component2;
static struct importedskeleton imported2 = {1, {
  {"console", &component2.console}}
};

static struct attributedskeleton attributed2 = {1, {
  {"position", &component2.position}}
};

struct exportedskeleton hwbase_hw = {
  {&_component, &hwbase_hw},
  3, {
    {"main", {&hwmain_mainmeth, &component2}},
    {"binding-controller", {&_bindingcontroller, &imported2}},
    {"attribute-controller", {&_attributecontroller, &attributed02}}
  }
};
```


Composite component ADL

- **No required code**
- **Describe**
  - Contents (which can be either primitive or composite component)
  - Content bindings
  - Content assigns
  - Controller
    - A static tool (for generate required code) and a dynamic shared part
    - Existing controllers
      - –Static - allows only static fixed configuration
      - –Dynamic - allows dynamic reconfiguration (adding, removing, changing)
      - –Secure - controls all interactions using a Reference Monitor

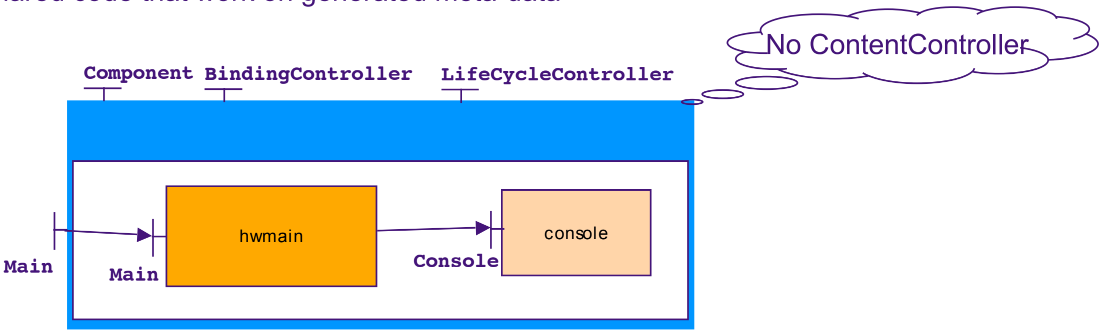



```
composite hwBase implements RootType {
      contains console = arm.sa1100.h3600.video.lib.screen
      contains hw = hwmain
      binds this.main to hw.main
      binds hw.console to console.console
      assigns hw.position = 3
      controller org.objectweb.think.controller.Static
}
```

Static controller

Aims

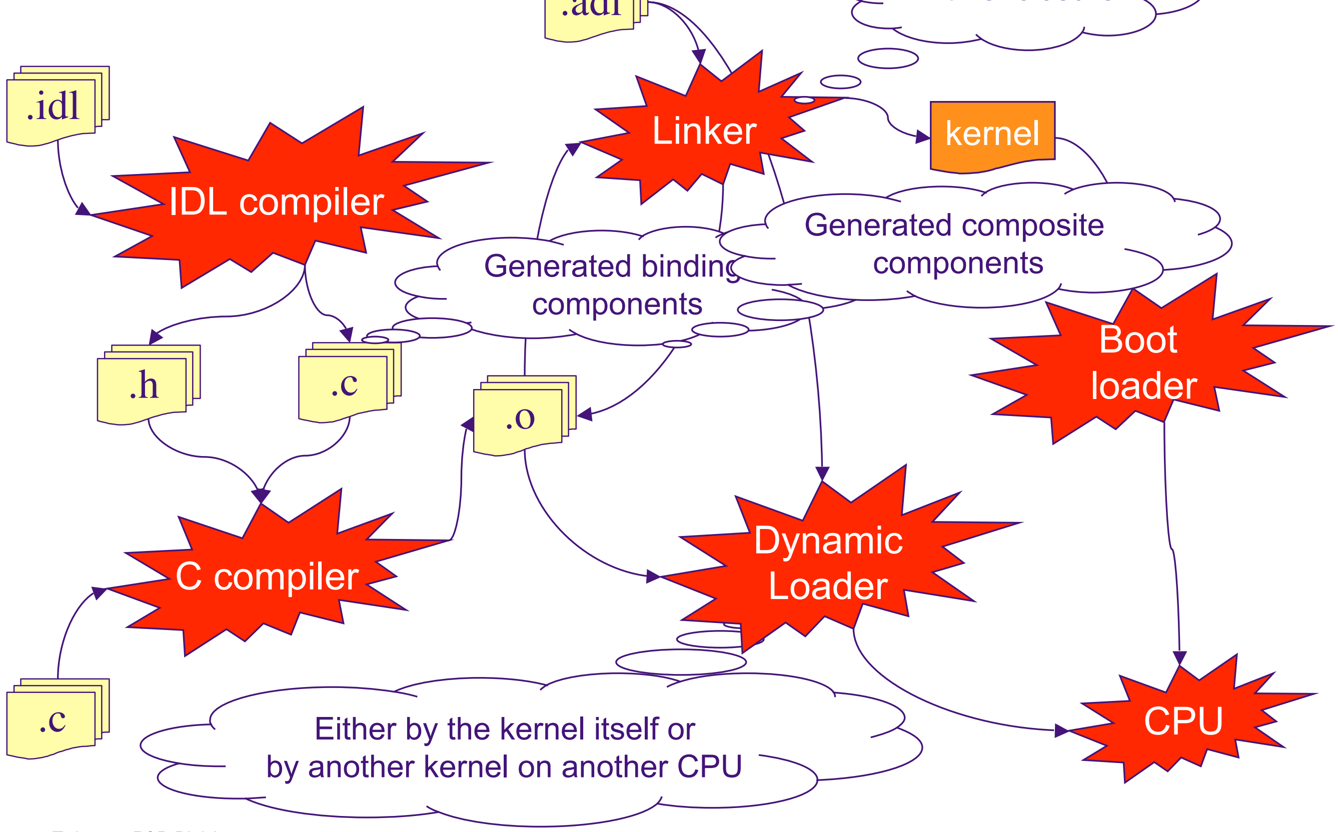
- Manages external bindings
  - No interception realized
- Creates all sub-components
  - This phase was done statically
- Binds sub-components together
  - By subsequent call to Component and BindingController of sub-components
    - A possible optimization was to fill statically dependencies on sub-component structures
- Assigns sub-components attributes
  - By calling AttributeController of sub-components
- Starts all sub-components
  - By traveling sub-components graph to respect order initialization
- Shared code that work on generated meta-data



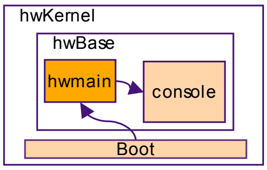
Generated meta-data : hwBase

```
static struct bxmlcomponent kc1[] = {
  {&hwBase_console.myreference, NeedStart},
  {&hwBase_hw.myreference, NeedStart},
};
static struct bxmlbinding kb1[] = {
  {1, 0xff, Mandatory, "main", "main"},
  {0, 1, Mandatory, "console", "console"},
};
static struct bxmlattribute ka1[] = {
  {1, "position", (void*)(3)},
};
static struct ecdata hwBase = {
  sizeof(kc1) / sizeof(kc1[0]), kc1,
  sizeof(kb1) / sizeof(kb1[0]), kb1,
  sizeof(ka1) / sizeof(ka1[0]), ka1,
  {&_ECci, &hwBase},
  {&_ECbc, &hwBase},
  {&_EClcc, &hwBase},
};
```

Deployments



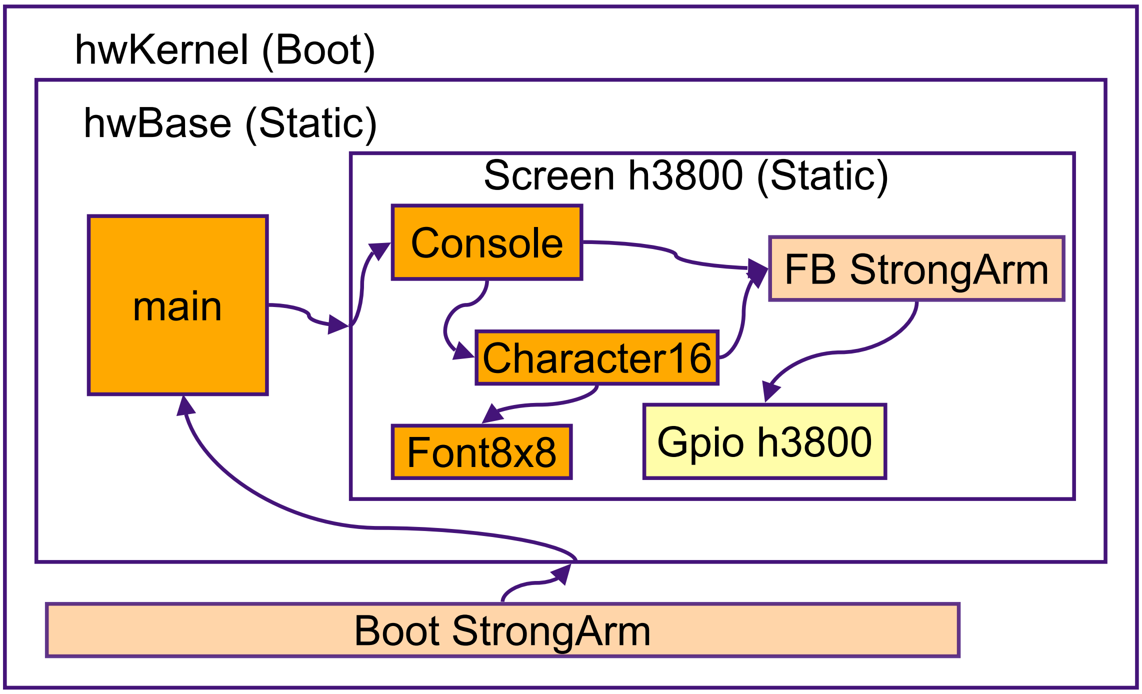
Final composition : `hwKernel`




```
composite hwh3600kernel {
      contains boot = arm.sa1100.boot.lib.boot
      contains main = hwBase
      controller org.objectweb.think.controller.Boot
}
```

```
composite hwh3900kernel {
      contains boot = arm.pxa.boot.lib.boot
      contains main = hwBase
      overloads main/console = arm.pxa.h3900.video.lib.screen
      controller org.objectweb.think.controller.Boot
}
```

"Hello World" for iPaq h3800



Overview

- **I - THINK and CBSE**
  - Mastering software complexity
  - Component-Based Software Engineering
  - Expected benefits
  - The THINK initiative
- **II - Fractal component model**
  - Concepts
  - Principles
  - Organisation (open component model)
  - APIs
- **III - THINK, a Fractal support for Operating System**
  - Interfaces
  - Components
  - Development
  - Configuration
  - Deployment
- **IV - The THINK component library**
  - The library
  - Supported hardware
  - Typical composition
  - Evaluations
  - Kernel constructions
- **V - Conclusion**
  - Summary
  - Links

The KORTEX library

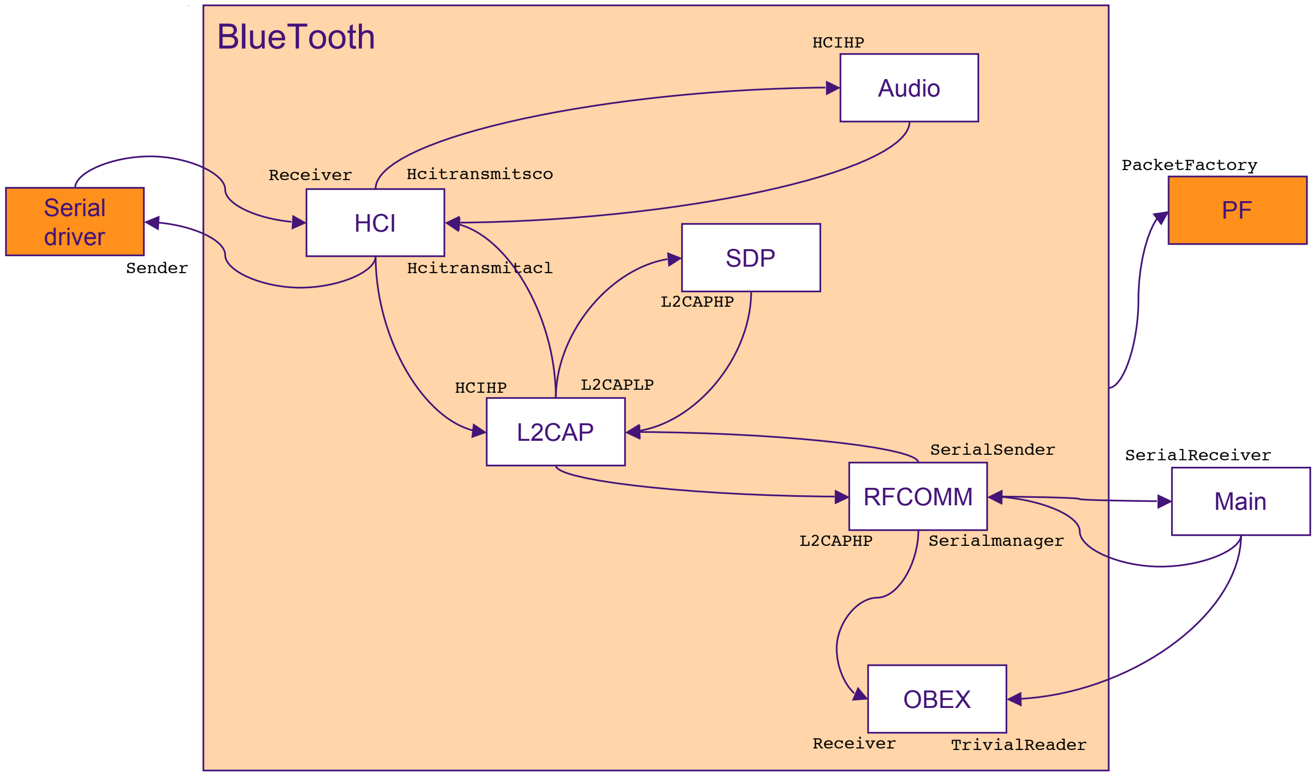
- **Hardware abstraction components (HAL)**
  - Boot
  - Exceptions (PIC, timer, …)
  - MMU (page table, TLB, exceptions, …)
  - Cache
  - Device drivers
    - Human devices: frame-buffer, keyboard, touch panel, …
    - Storage decides: Disk, Flash
    - Communication devices: Serial port, Ethernet, Bluetooth
- **Operating system services**
  - Memory management components (flat & paged memory)
  - Thread and scheduler components (cooperative, round-robin, priority, QoS)
  - Network components (x-kernel)
    - Ethernet, IP, TCP, UDP, SunRPC
  - Bluetooth components
    - HCI, L2CAP, RfComm, HDI, Obex
  - File system components (VFS API)
    - JFFS2, E2FS, NFS, Ramdisk
  - Binding components (local & remote communication)
  - Service components (process, dynamic linker/loader, trader, ...)
  - Posix components (for application portability)

Supported Hardware

- **Conform with Fractal components model**
  - PowerPC G3 / G4
    - Apple Power Macintoshes (Only New-world architecture)
  - ARM
    - Intel StrongARM (HP/Compaq h3600, h3800)
    - Intel xScale (HP/Compaq h3900, h2200, h5500)
    - Portal Player's PP5002 (Apple iPod)
    - Motorola Dragon Ball MX 1
- **Experimentations (past and future)**
  - PC (Intel x86) [INRIA]
  - Lego RCX (Hitachi H8 16Mhz, ROM 16KB, RAM 32KB) [INRIA]
  - Tini Internet Interface (microcontroller DS80C390, ROM 512KB, RAM 1MB) [INRIA]
  - DSP [ST Microelectronics]
- **Simulator**
  - HAL components built on Unix (Linux, Mac OS X, Cygwin)
  - Run Think kernels on Unix processes
    - Porting and debugging purpose

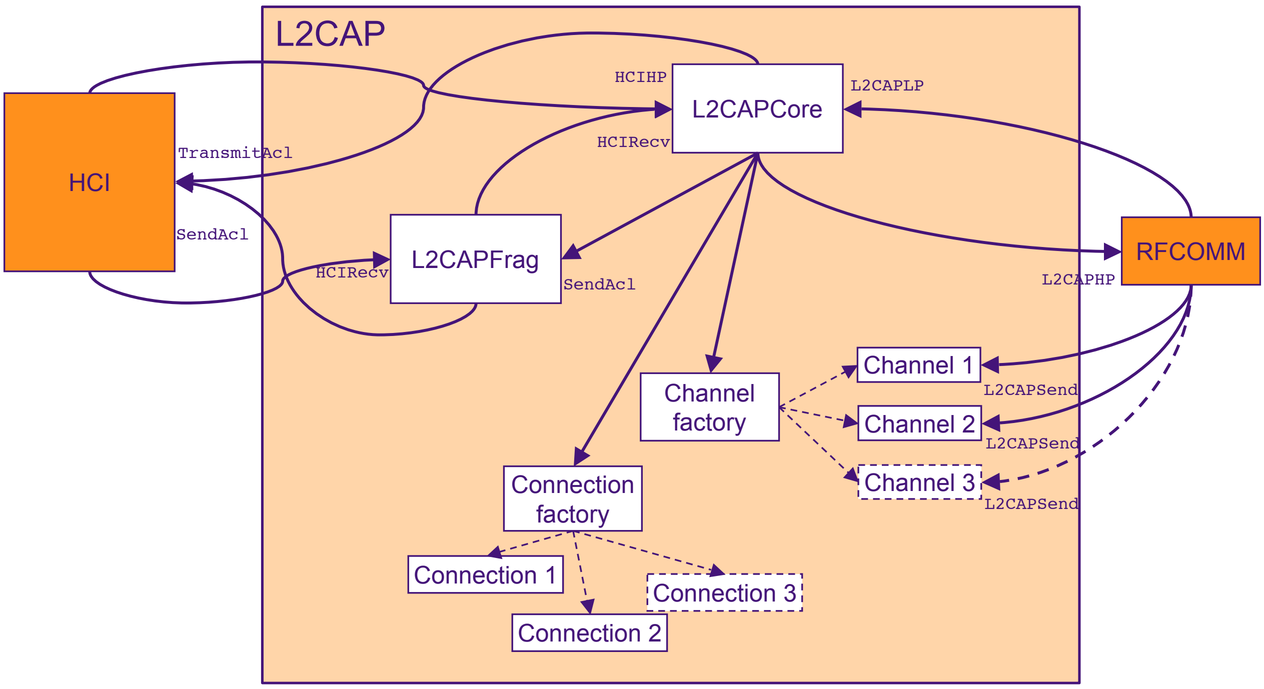
Bluetooth (1)

[Anthonny Pellerin]



Bluetooth (2)

[Anthonny Pellerin]



Scheduling interfaces

```
interface Scheduler {
      void  initJob(Job job,
                    byte* function,
                    byte* stackbase,
                    int stacksize,
                    int arg);
      void  destroyJob();
      void  addJob(Job job);
      void  removeJob(Job job);
      Job   getJob();
      void  yield();
}
```

```
interface Semaphore {
      void  P();
      int   tryP();
      void  V();
}
```

```
interface Mutex {
      void  lock();
      int   trylock();
      void  unlock ();
}
```

```
interface Job {
      void      run();
      void      enQueue (Job* jobqueue);
      void      deQueue ();
      Job       getQueue ();
      byte*     getContext();
      void      activate();
      MMUSpace  getSpace();
}
```


Component sizes

	Sub components	Size (KB)
Limited TCP/IP stack	9	**15.5**
Console & graphic drawer	9	**9**
H3900 frame buffer (including video memory)	1	**160**
H2200 frame buffer (video memory on controller)	1	**0.9**
Preemptive priority scheduler & semaphore	11	**5.7**
ARM cache & flat memory management	7	**3.9**
Think controllers	4	**3.1**
C runtime (sprintf, str_{xxx}, mem_{xxx}, divsi3, …)	n.a.	**7.7**
ARM EFL dynamic components loader	4	**2.2**
JFFS2 flash reader	3	**9.1**
PXA serial port & ymodem	2	**1.7**
H3900 touch panel	5	**3.9**
Bluetooth stack (HDI, RFCOMM, OBEX, …)	17	**83**

Performance overheads

- **After configuration, only interface calls add overhead**
- **Interface calling overhead**

```
itf->meth->foo(itf->data);
Void fooimpl(void* _this) {}
```

- ARM

	Instruction
Interface calling	5
Procedure calling (with context)	3
Procedure calling (without context)	2

- PowerPC

	Instructions	Time	Cycles
Interface calling	6	0.016 µs	8

Memory footprint overheads

- **Primitive components**

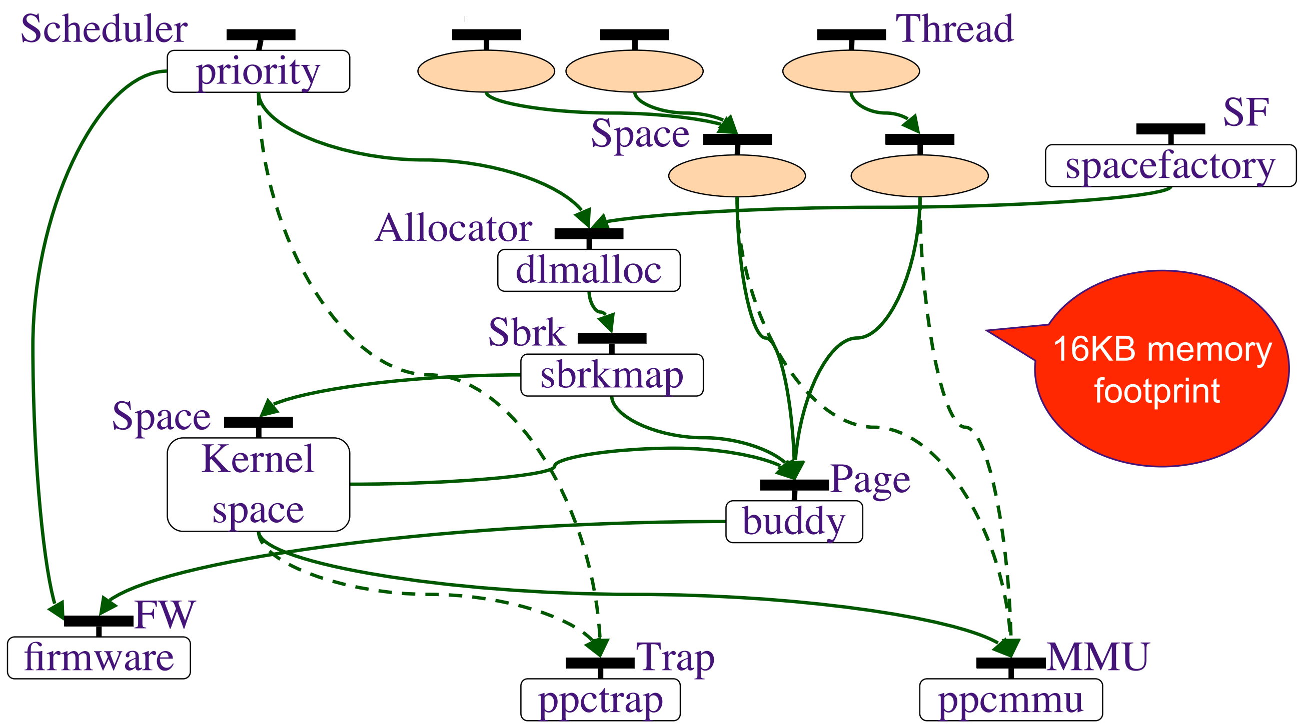
	Delegation overheads (bytes)	Optimized controller (bytes)
Exported interfaces	4 + 12 * n	8 * n
Imported interfaces	4 + 8 * n	0
Attributes	4 + 8 * n	0

  - For the 3KB IP component, overhead is about 2%,
  - For a 1 KB frame buffer, overhead is about 6%.
- **Composite components**

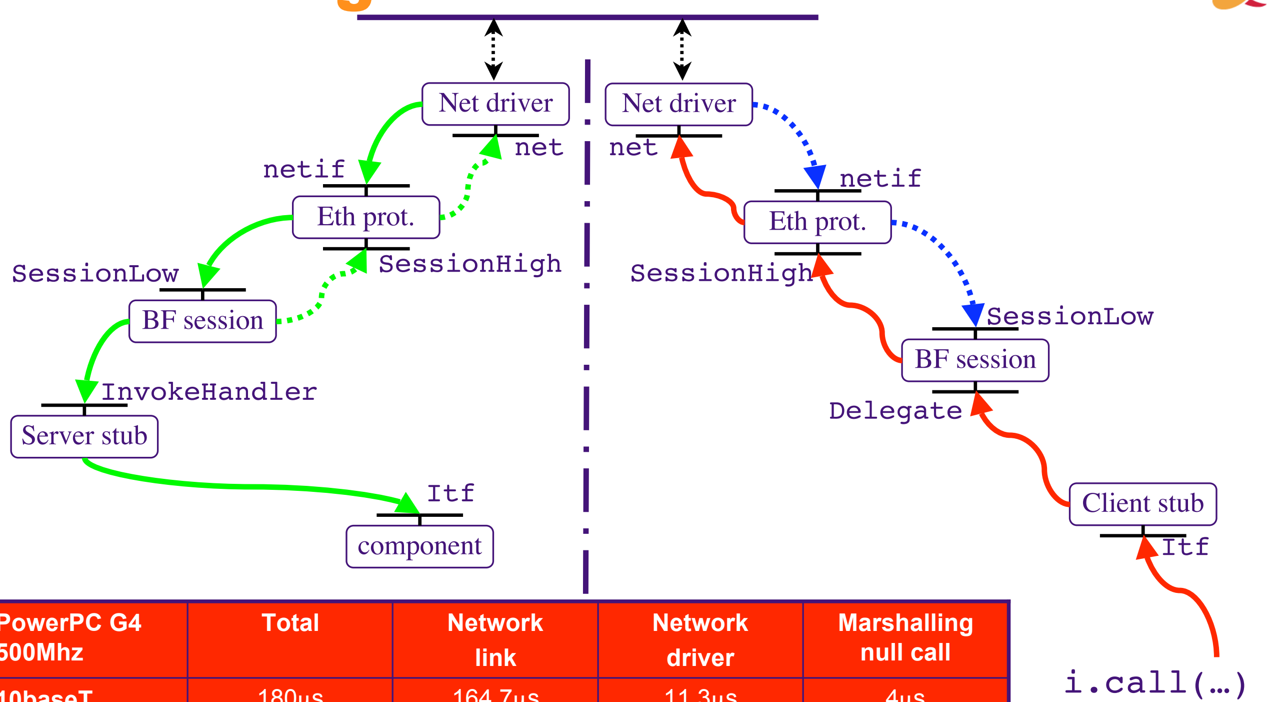
	Standard controller (bytes)	Optimized controller (bytes)
Sub components	4 + 8 * n	0
Sub bindings	4 + 12 * n	0
Attribute assigns	4 + 12 * n	0

  - For the TCP/IP stack with 10 sub-components and 25 sub-bindings, overhead is near 400bytes.

Building a simple µ-kernel

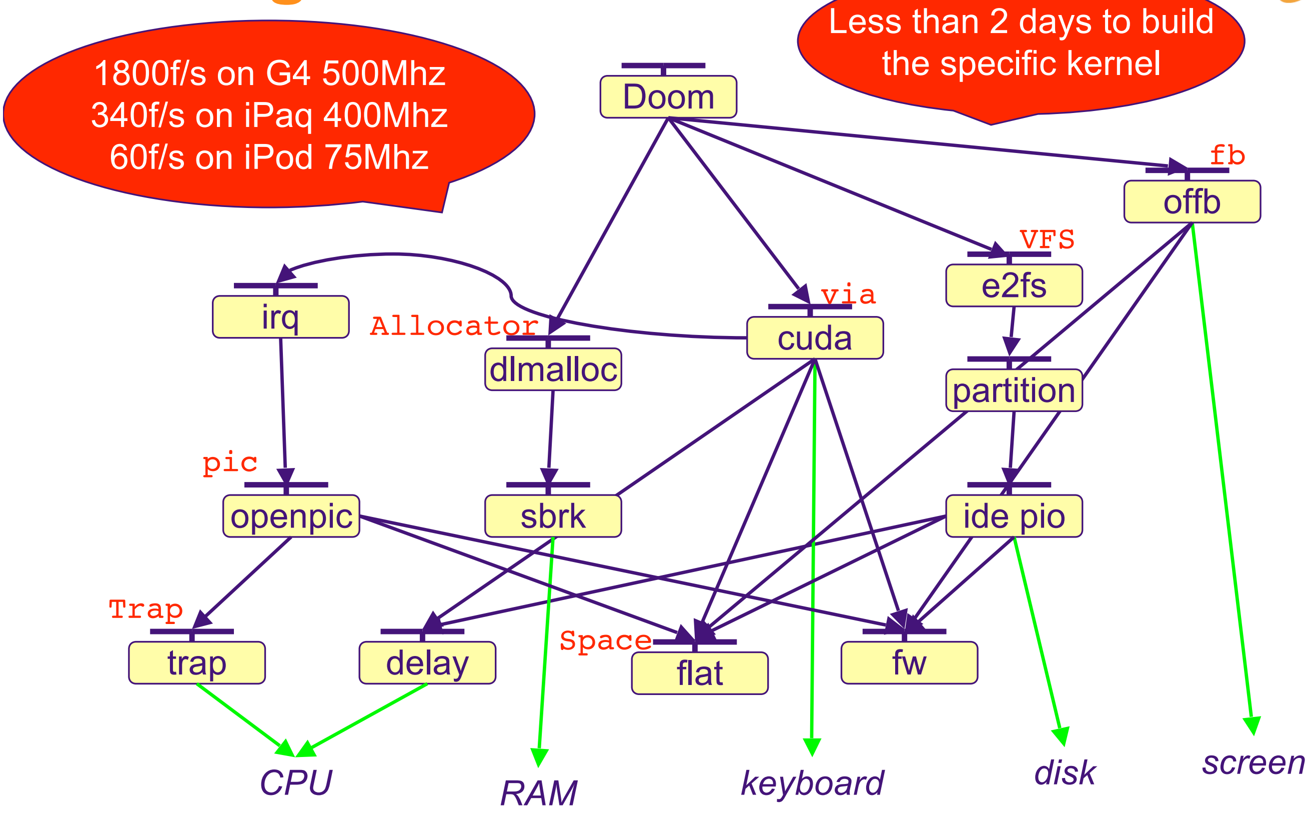


RPC binding



PowerPC G4 500Mhz	Total	Network link	Network driver	Marshalling null call
10baseT	180µs	164.7µs (91.5%)	11.3µs (6.3%)	4µs (2.2%)
100baseT	40µs	24.7µs (61.7%)	11.3µs (28.3%)	4µs (10%)

Building a Doom kernel



Kernel construction illustrations

- **Plan-P application specific kernel**
  - Programming Language for Active Networks
  - Ethernet Bridge evaluation
    - 86.7 Mbits throughput on Kortex-based kernel
    - 65.0 Mbits throughput on Linux
- **Kaffe**
  - Looks like a JavaOS
  - 1.6 MB memory footprint (125KB Kortex, 475KB Kaffe)
- **Secure kernel (Mohamed-Tahar JARBOUI)**
  - By defining a secure composite controller to checks interactions
- **QoS kernel (Jean-Charles TOURNIER)**
  - By defining resource managers that provide QoS

Overview

Conclusion

- **Effectiveness of component model for building component-based operating system**
  - "Component-based" does not mean "less efficient"
    - Interface cost == C++ virtual cost
  - Operating system development simplification
    - Components and interfaces clearly identified
  - Quick system deployment
    - “Lego-like” construction of software by assembling pieces a.k.a. components
  - Memory footprint reduction and performance increase
    - Removing unused services
    - Fine resources access allowed
- **Yes, that works**

Links

- **Think home page**
  - http://think.objectweb.org
- **Fractal home page**
  - http://fractal.objectweb.org
- **Questions ?**