Introduction to Fractal systems

Victor Andritoiu, Laurent Carcagno

Introduction to Fractal systems

Managing complexity for embedded systems

V. Andritoiu, OpenTekhnia
L. Carcagno, Techniware

v1.1 - December, 2012

Summary

Each time that a new technology is made available, each time that a new component pushes the edge of electronic systems, we have to face a new danger: complexity, in its common meaning. If for a simple desktop computer or for an internet tablet, complexity means bugs, at the level of the planes' or cars' or trains' embedded systems, it means possible critical failures. This subject is not new and hordes of engineers work on it, but currently system design is either all about long term research, or about patching current industrial products. One would say «yet another design paradigm», others would talk about unpractical concepts, but we are talking about what is possible to do, what should be done as a possible way of designing systems and what we are already working with.
We have started from one single and, in some way, small distinct subject: simulation for testing, integration and validation of complex systems. We have developed one solution (Open System Simulation Solution) that allows us to implement test beds for embedded systems. And we realized that we have in our hands a complex embedded system that has what we call a fractal architecture, can be auto-testable and could be safety critical compliant with some efforts. But what means precisely, at least from our perspective, «fractal architecture» ?
A system provides services, but in order to do so, most of the time it is based itself on one or several subsystems providing more basic services, like for example communicating. A fractal architecture system will be a system that is able to provide the same basic and mandatory services at all of its scales. For example, we consider as one of the mandatory services «communicating and manipulating logical objects that can have a physical representation». In other words, it is about working with structures of data that can be, if necessary, obtained from physical inputs or that can represent the state of physical outputs.
Our purpose is to demonstrate that artificial systems based on a fractal architecture are one of the best approaches for building complex embedded systems. Moreover, they could be interoperable in an interconnected, but unstandardized world.

Table of contents

A. Mandatory services for a complex system

1. Interacting with the environment

Whatever the provided high level service is, if we consider a system as a black box, the most basic thing that it should do, is to be able to interact with its environment. As we consider here embedded systems (so computer based systems, but considered only for their electronic part), we would say that this interaction can be done thanks to:
- sensors
- actuators (including display devices)
- DIO
- AIO
- any kind of communication bus.

2. Communicating

We are talking about systems, so we mean, first, a set of subsystems that necessarily communicate each other.
From our perspective, this internal communication should not be different from external one, in that meaning that a system should be interoperable and open. In other words, even if some specific interfaces would be necessary, the same mechanism has to be put in place everywhere.

It is interesting to define more precisely what communication means.
When we started studying the fractal systems concept, we did it a bit (and maybe a lot) thinking to how nature itself is build. Biomimetics should not be used only for mechanics...
Moreover, as we consider that systems should be interoperable with their environment (external systems), we have to imagine some communication way close to the human one. Of course, we are not looking for some kind of natural language. We are too far from the necessary technological level for this and, maybe, we will never be closer simply because, even between humans, natural language introduces ambiguous understanding, which is unacceptable for deterministic systems. But we talk about objects. Object oriented programming has been a major step in computer engineering, but more than this, it has been a way to apply human mind constructions to computers.

We consider that a system, today and tomorrow, has to work (compute) with objects. These objects can be primitive (scalars, for example) or complex (structures), with or without a related behavior. However, when a human gives a car key to another one, he does not specify what to do with the key (at least nine times over ten). In the same way, two systems will not proceed to data exchange containing behavior information. At least not in our first level of fractal systems design. It means that systems would be able to communicate (cooperate) only if they have a common data model.

So...
Another basic and essential system service will be «communicating data objects».

3. Scheduling

Nothing can be done without scheduling. This is true in the real life and it is true for systems. However, in our understanding, scheduling contains event management concepts. Indeed, a system will not schedule a task only as per a time trigger, but as per any kind of event.
Scheduling mechanism for tasks is the third essential service.

4. Tasks

And if we are able to schedule a task, we must be able to build a task as a set of actions that defines a functional behavior. We have to understand here the word «task» in its common meaning. From the software point of view, we consider that this will correspond more or less to an application. Then, a fourth basic service would be «scheduling applications and applications' components».

5. Services, from a software point of view

There is one typical software approach that is certainly the best way to describe, at least to a software engineer, what is the expression of the previous described services. We talk about the API (Application Programming Interface).

At this level we will start naming our stuff Fractal Systems Core (FSCore). Indeed, the purpose of our document is not only to describe what a Fractal System could be, but mainly how it would work and which is the basic core to be defined in order to provide all the necessary services for such a system.

Let's consider the services as defined above:

1. environment interaction: the FSCore cannot reasonably provide services for ANY kind of I/O. However, it should provide services for defining them. From this perspective we can imagine the following methods (at this stage, we are at a very high level of definition and methods are very far from any implementation; however, we will use C++ pseudo-code to do so):

```
class FSCore {
...
public:
      void addAgentLogicalAdapter(const FSAgentAdapter& adapter);
      void removeAgentLogicalAdapter(const std::string& name);
      void addAgentPhysicalAdapter(const FSAgentAdapter& adapter,
                                   FSDevice& dev);
      void removeAgentPhysicalAdapter(const std::string& name);
...
};

class FSApp {
      void asynch_send(const FSAgent& dest, const FSObject& msg);
      bool synch_send(const FSAgent& dest, const FSObject& msg);
      void asynch_receive(const FSAgent& src);
      void synch_receive(const FSAgent& src, FSObject& msg);
};
```

2. communicating. If the environment interaction allows the system a physical send or reception of data, the way that it uses this data is related to the communication layer defined below:

```
class FSObject {
...
public:
      std::string name();
      setName(const std::string& name);
      void set(const std::string& path, const FSVariant& value);
      FSVariant get(const std::string& path);
      long long timestamp();
      void setTimestamp(long long t);
      std::string type();
      void setType(const std::string& type);
...
};
```

3. scheduling and tasks: we consider that any FSCore application can be scheduled and, then, it is in some way a task within the distributed system. This means that we will add new methods to the previous definition of *FSApp*, as following:

```
class FSApp {
...
public:
      typedef enum {...} AppState;

      void init();
      void run();
      void terminate();

      AppState state();
...
};
```

Then we have three different phases for the app, while the run phase can be synchronous or asynchronous, cyclical, or not. On the other hand, we will have:

```
class FSCore {
...
public:
      FSApp* installApp(const FSAppPackage& package);
      bool uninstallApp(FSApp& app);
      bool startApp(FSApp& app);
      bool pauseApp(FSApp& app);
      bool resumeApp(FSApp& app);
      bool stopApp(FSApp& app);
      bool startAppInstance(FSApp& app, FSAppInstance& instance);
      bool pauseAppInstance(FSApp& app, FSAppInstance& instance);
      bool resumeAppInstance(FSApp& app, FSAppInstance& instance);
      bool stopAppInstance(FSApp& app, FSAppInstance& instance);
...
};
```

Of course, we need to take into account that FSCore's instances would be local to a system's node (since we consider distributed systems). Then, we will eventually have one entry point level class that is a meta-controller of the whole system. This could look like:

```
class FSCoreAdapter {
...
public:
      FSApp* installApp(const std::string& name, const FSAppPackage& package,
            FSNode& node);
      bool uninstallApp(const std::string& name, FSApp& app, FSNode& node);
      bool startApp(const std::string& name, FSApp& app, FSNode& node);
      bool pauseApp(const std::string& name, FSApp& app, FSNode& node);
      bool resumeApp(const std::string& name, FSApp& app, FSNode& node);
      bool stopApp(const std::string& name, FSApp& app, FSNode& node);
      bool startAppInstance(const std::string& name, FSApp& app,
                              FSAppInstance& instance, FSNode& node);
      bool pauseAppInstance(const std::string& name, FSApp& app,
                              FSAppInstance& instance, FSNode& node);
      bool resumeAppInstance(const std::string& name, FSApp& app,
                              FSAppInstance& instance, FSNode& node);
      bool stopAppInstance(const std::string& name, FSApp& app,
                              FSAppInstance& instance, FSNode& node);
...
};
```

Related to the communication aspect, we need to introduce as well the asynchronous aspects of scheduling. For example, has an app to process messages only in a cyclical way ? Of course not. This means that a trigger mechanism has to exist. Then, the *FSObject* definition has to be improved, adding following methods:

```
class FSObject {
...
public:
      void registerCallback(FSObjectCallback* callback);
      void unregisterCallback(FSObjectCallback* callback);
...
};
```

After this rough introduction to a minimal API, we need to clarify few concepts.

Let's start with the most confusing one: the «*Agent*», represented here as a *FSAgent* class. This represents any external entity with which the concerned system is supposed to interact. Of course, *FSAgent* class is an internal representation of that entity.

Another issue could be the notion of «type» of a message.
Indeed, we said that the system should be able to deal with primitive or complex data. This means that we need to declare in some way that type and the way that it is used depends on it. In a more practical way, a C++ implementation means that we need to store a complex structure as a kind of a dictionary tree, since we are note supposed, of course, to know all the types by the time of the compilation. This is because those types will depend on the apps, while the apps can be loaded/unloaded dynamically.

On the other hand, the meaning of system's node has to also to be clarified and detailed. This will be done in the next chapter, related to the architecture.

B. Fractal functional architecture

1. Delegation

When we look at our own human body, we quickly understand that notion of delegation.

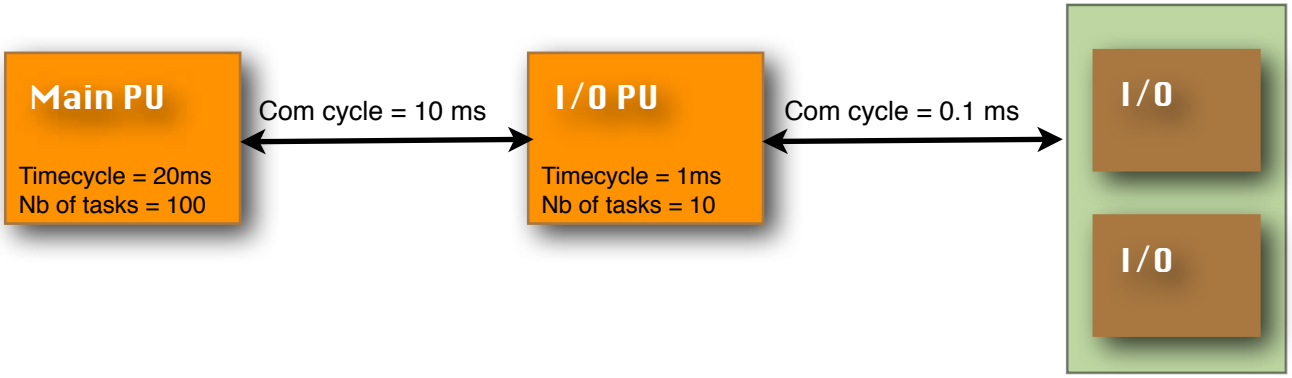
Move you hand ! Your brain takes care. There are some neural aspects implemented in the hand itself, but the hand is controlled by the brain. This is one kind of delegation, at least logically speaking. Instead of having a brain everywhere, each «component» of our body takes advantage of the presence of one powerful computation unit, our brain. Then, when about how to move, we can say either that the brain asks to the hand to move that or that way, or that the hand asks for what to do to the brain.

One would say that this explanation has no meaning from the nervous system point of view. Maybe... However, this is not the point. The point is that, if we had to implement a robot arm, we will not locate a computation unit at the arm level. Eventually, we will put there only some power electronics that manage the actuators.

Another typical delegation situation is the so fashioned cloud computing. There are dozens of different cases, but one of them is more understandable from the embedded systems point of view. We are talking about the high performance computing, implemented in a «cloud» context. This starts from the idea that it is definitively faster for your one single computer unit to send a computation request and get the result than doing it locally. Of course, this will start being true from one specific threshold of computation workload and given an adapted remote computation capacity. PiCloud (http://www.picloud.com/) is a good example, but dozens of equivalents exist now.

In the case of Fractal Systems, the idea is very similar. Here is about available computation workload, but also available actuators or sensors. The delegation is bidirectional, from the highest computation capability to the lower one, and vice versa, but also between two equivalent capabilities.

Let's first consider a simple architecture, not taking care about the fact that is a fractal system or not:



We could have the following:
- Main PU: main processing unit, implemented as a desktop or industrial PC based on Core i7, RAM 16Gb etc etc.
- I/O PU: inputs/outputs processing unit, implemented as an embedded ARM computer based on ARM v7, RAM 512 Kb etc. etc.
- I/O: digital or analog inputs/outputs.

This very classical situation, without being a Fractal System, uses most of the time the delegation mechanism.
Indeed, we can imagine that the Main PU is processing some high level optimization algorithms, very CPU grasping, while the I/O PU has to update in a very fast way the physical I/Os. From the point of view of the whole system, Main PU is delegating to the I/O PU what it is not able to do: interacting physically with external world. This could be done sending and receiving updates through simple messaging, each 10ms, while the I/O PU has to update the I/Os much more often than this. At the same time, the I/O PU would not be able to compute any high level algorithm, which is «delegated» to the Main PU.

This case is a very common delegation: hardware capabilities are used as per the needs. However, it would be true for distributed embedded systems, and not for single CPU ones.

Let's see, in the next section, how the Fractal Systems are changing this basic case.

2. Fractal architecture

First, we need to define Fractal Systems architecture:

«Fractal Systems software architecture is a scale free distributed architecture providing any available service at any available level.»

We must pay attention to one word there: «software».
Indeed, one of the main purposes of the Fractal Systems is to be hardware agnostic. This means that we should not care about hardware platform or Operating Systems. We must only focus on the Fractal System services, as software services. Of course, this does not mean that we are not able to access real world (through I/O physical signals)., but only that we manipulate data at software level, without taking care about how those data are transformed to and from signals.

Let's come back to our previous section's simple situation and let's imagine a particular use of this.

The simple system described above could be a robot.

From the functional point of view (ignoring OS level), the Main PU could have to deal with the following tasks:
- communication task;
- process I/Os received from the I/O PU. The inputs would correspond to sensors detecting environment obstacles or even 2D or 3D self position, battery charge, etc., while the outputs would be motors control;

- compute the best trajectory for given inputs processing and maintain home distance in accordance with battery charge;
- high level decision task (for example, do I have to go back to home for battery charge, do I have to explore that zone, etc.);
- emergency task: best trajectory to home;
- remote control task: manages communication with a remote command center.

All these tasks have not to run, but could be started on demand and then, scheduled. The reason could be the fact that each one needs energy and there is no reason to waste it.

The I/O PU could need to manage the following tasks:
- communication task;
- read/write cyclically the I/Os;
- execute emergency brake in case of immediate obstacle detection (imminent collision);
- battery charge supervision (could ask for the Main PU emergency task to be executed if battery charge is too low).

In a classic system, some kind of messaging could used in order to manage delegation, requests, data exchange etc.
For example, we can imagine that in case of battery charge level too low, the I/O PU would send a message to the Main PU which would trigger the emergency task execution.

In the case of a Fractal System, things are different.
Indeed, we said that all the services would be available for everybody (at least any piece of hardware that is able to execute a piece of software). This means that the I/O PU would have an access to the previously defined *FSCoreAdapter*. Then, instead of sending a message to the Main PU, it would just do something like:

```
fsCoreAdapter.startApp('mainpu', emergencyApp, node);
```

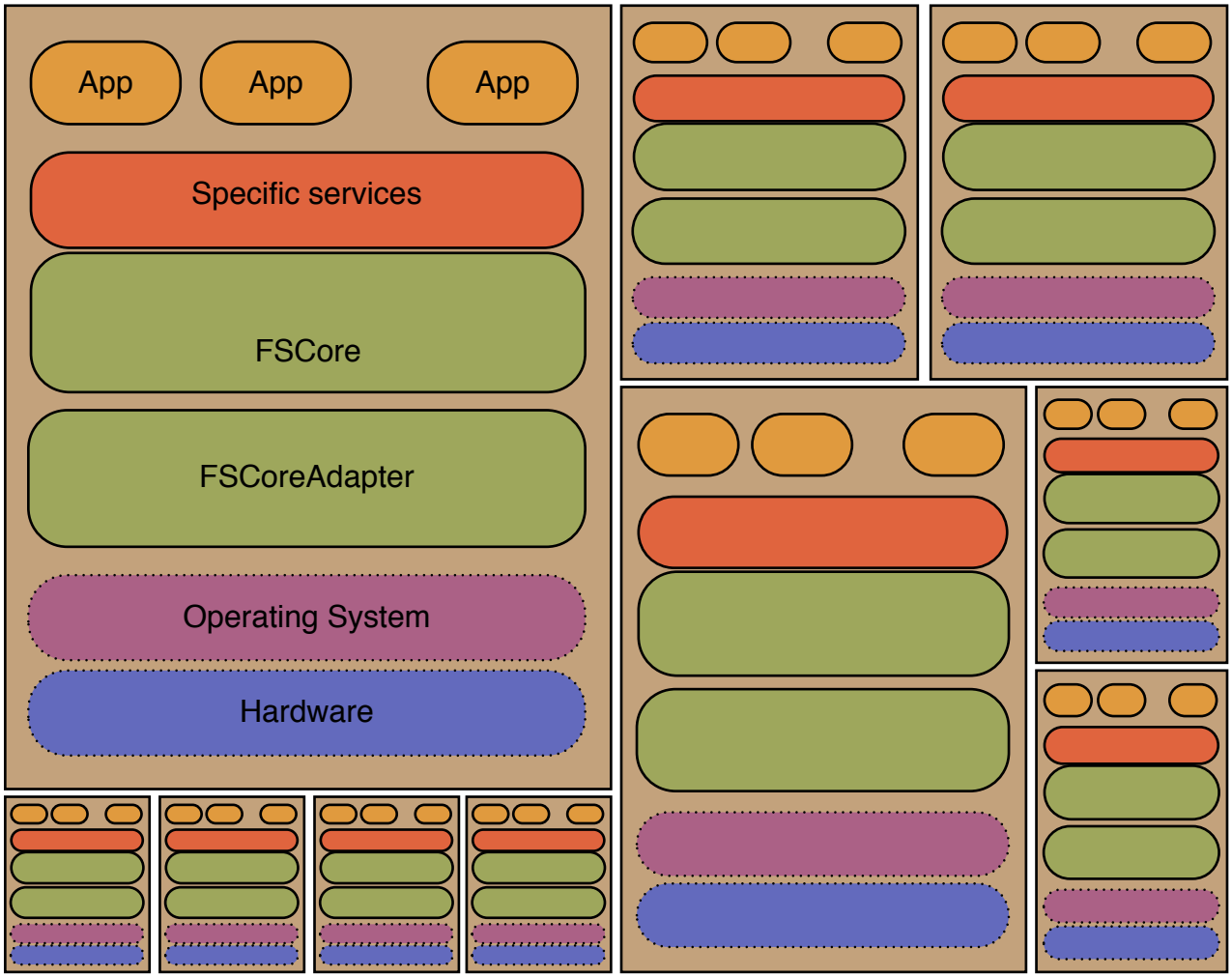
Notice one thing: this instruction would be called within an *FSApp* running on the *'iopu'* node (corresponding to the I/O PU component), which means that all the *FSCore* services are available inside any *FSApp.* Then, theoretically, we can logically nest applications in an infinite way, precisely like for a fractal object.

However, this would not be the only way to execute the action.
In our definition and understanding of what Fractal Systems should be, we have a data distribution mechanism that is giving the possibility to share with everybody a given object. Like for example a simple boolean that could be used as a trigger for the emergency task at the Main PU level...

The two mechanisms are slightly different: the first one is certainly asynchronous, while the second one is synchronized by the data distribution mechanism. Indeed, we can hardly imagining other way to implement data distribution than the one of a cyclical update. Then, distribution cycle must be adapted to the realtime constraints.

However, at least in our case, we already can see that the implementation of a distributed system based on data distribution is not an obstacle to realtime constraints, since every case would have a solution provided by the Fractal Systems services.

Let's try now a first visualization of what a fractal Architecture could be.



It is assumed that:
- *FSCore* and *FSCoreAdapter* layers are OS agnostic (ideally fully portable);
- a specific service is supported by the *FSCore* layer, but also by the OS;
- an application is supported by the *FSCore* and but it is not supposed to be independent of the OS, since the application implements a specific behavior and eventually uses specific services;
- the communication between components (brown rectangles) is done thanks to realtime synchronous data distribution and synchronous or asynchronous remote calls.

The size of the brown rectangles wants to provide an information about the scale of the component. By scale of the component we measure not only a physical scale in terms of «embeddability», but also a computation workload capability. We suppose, as well, that the more a component is embeddable, the less its computation capability is important.

On the other hand, here we've got the «node» meaning. It corresponds eventually to a component, but more than this, it is a local runtime that provides a full set of FSCore services.

3. Determinism

One could immediately ask a very tough question: could such a system be fully deterministic ? And the hidden question there, is: could such a system be safety ?

Well, the answer is the same as for any existing distributed system.
The domain that we (writers) know the best, it is the one of railways. Currently, the most advanced railways systems are communication based distributed realtime ones. They are asynchronous, based on messaging and quite heterogenous in terms of equipments, even if efforts are done for decreasing this factor (heterogeneity). You will find there powerful x86 servers, as well as embedded units based on Power PC, ARM SOC (System On Chip), Linux, Windows, as well as QNX or VxWorks operating systems. Nobody is asking, however, the magic question: COULD such systems be safety ? The reason for not asking is certainly the fact that most of us already traveled by metro in some city in the world and we are all still alive. Few accidents are known to be a consequence of a system failure.

What we want to say here is that Fractal Systems as defined by us cannot be excluded from the safety critical systems only because they are a new concept, a new architecture. If someone spends time and money to develop such a system, it is 100% certain that it would be possible to provide proofs for a deterministic behavior, at least at the same level as existing systems in railways, or any other domain. It is all about processes, methods, testing etc.

In fact, we definitely think that Fractal Systems are even a better approach for critical systems. The main reason is that Fractal Systems provide a huge simplification to the functional level, splitting mandatory basic services and highly abstracted functional behavior.

Let's start with something that is not only related to Fractal Systems, but to any data distribution enabled system. We talk about data synchronization or replication.

Independently of the function itself, a critical system using such a data distribution middleware would have to provide provide the proof that at any time a given distributed value is the good value to be used by everybody. This is about not losing data, having a deterministic updating behavior and managing, eventually, data quality that a consumer can check. Once this done, the way that functional part deals with the data, is another story, but a significative part of the safety demonstration is done.

In the same way, providing the proof of a deterministic behavior of each *FSCore* service, is certainly not enough to have a deterministic system. but it is a huge step that has to be done only once.

Then, Fractal Systems architecture and related middleware will not affect the capability to certify a system with safety critical compliancy. In the best case, this will even ease that process, providing one shot work, whatever the specific implementation is.

The only real problem (but this is true for any chosen technology) is to be able to do so for all the targets. Fractal Systems suppose hardware fragmentation. It is clear that for a safety critical system, this fragmentation has to be reduced as much as possible.

4. ... or no determinism

In opposition with the deterministic concept, we have scripting and code interpretation.
In order to build a safety critical system, we need to compile. This is the current situation and we will not try to go out of the existing bounds. Then, adding scripting features to Fractal Systems could appear as incredibly counterproductive.

However, for a given complex system, the experience shows to us that only some components are safety critical. The interests in adding scripting facilities are numerous:
- flexibility and adaptability in the system use;
- possibility to provide a programmatic «non-programming» interface to the user: indeed, even if scripting is programming, this is the easiest way to do so, the one that could be the closest to the human natural language, keeping an unambiguous approach;
- possibility to provide, very easily, a lot of new features to the system (the typical example is Python, that provides an infinity of libraries easy to use, when Python interpreter is embedded to an app);
- more generally, capability to extend the system.

Even if this should not be a mandatory feature of Fractal Systems, the computation power's year to year improvements are giving to us an additional argument to introduce it in the core specifications.

What does this mean ?
First of all, all the *FSCore* and *FSCoreAdapter* features should be available through the scripting language(s). The consequence of this is that the corresponding interpreter should be available at any scale of the system. This is the only limit to the feature acceptance.

5. Interoperability

Communication through data objects is simple. In some way, we have been inspired by the Web technologies and more specifically by the JSON format.
JSON format is a simple representation of a Javascript object, certainly one of the simplest when talking about human readable information. Web technologies have been using standard object representation from years and years. JSON is just one of the latest, one that is raising and replacing step by step, site by site, the XML one.
One of the most interesting aspects of JSON format (and certainly one that helps it spreading out quickly) is the fact that is quite easy to adapt to any programming language. Having a C++ object server side and sending data to a Javascript client, is just easy. This is a kind of communication interoperability.

We do think that this kind of formats can be adapted to Fractal Systems. We could talk more precisely about BSON (binary JSON, used by MongoDB engine - http://www.mongodb.org/), but any other similar to it could be good. Binary formats are, of course, more compact and better adapted to any realtime constraints. In the case of a binary format used for data distribution, it would be quite easy and natural to have adapters to external systems that convert external data representation to some *FSObject* instances.
In fact, those adapters would certainly do more than this, providing a data model conversion as well. But this is related to the specific functional implementations, and not to the raw data interoperability.

Of course, interoperability is not only data communication, but also functional requests (as services access). From this perspective, Fractal Systems should provide any core functionality as a standard remotely callable operation. We could imagine using WebServices, but the question is still open, since it is certainly possible to do better when talking about embedded systems.

C. From theory to realization

1. Implementation technology constraints

The first step in choosing the necessary technologies, is to clarify the equipments targets of Fractal Systems.
Given the «scale free» aspect, and dividing the problem into software and hardware platform aspects, we could have the following:

- software targets:
    - wide-spread operating systems, realtime or not, which means Linux, Mac OS X, Windows, QNX, VxWorks, PikeOS, Integrity, Solaris, etc
    - firmwares: Fractal System services should be implemented as well on physical targets without an Operating System.
    - FPGA: we consider this as the «lowest scale» of Fractal Systems. This is also a hardware target, but here it means that implementing Fractal Systems services is done without any other low level services than libraries dedicated to FPGA. In some way, could be an equivalent of the firmware case.
- hardware targets:
    - any server
    - any PC/Mac/Whatever computer
    - embedded boards (ex: SOC boards)
    - any processor architecture (x86, ARM etc)

As we can see, this is a very large set of devices. We can frankly say that we target any electronic device. And the consequences are a huge set of constraints on the selected technology.

Certainly, we can imagine to specifically implement the desired services to each needed target, project by project, subject by subject. But this would mean that Fractal Systems are just a cloudy concept that remains theoretical. Our purpose, but also experience, shows that we need to do much more.

Indeed, we consider that the software part is the real added value of systems development. At least, this could be the future. In that case, software development effort must be controlled, well defined, stable and «deterministic». This leads to the necessity to implement Fractal Systems services in such a way that very few variants would exist. Ideally, just one.

In a more practical meaning, this should be realized through one single version of source code.
Our experience of O3S development, which is our first implementation of a Fractal System, shows that it is possible, at least, to have the following situation:
- one single source code for: Linux, Mac OS X, Windows, QNX, Solaris, VxWorks etc. for both x86 and ARM architectures
- 90% of the previous source code common with firmware and FPGA targets, with an additional part that represents the low level services equivalent of a classical OS.

However, this excludes the scripting part. Indeed, we did not succeed in having this feature available on all the targeted platforms.

2. Language, frameworks and technical bricks

a. Similar architectures state of art

Fractal Systems are not introducing new concepts from the point of view of the data exchange, in that meaning that exchanging dictionaries of data is something already in use for several different system architectures, in the embedded world as well as in the cloud services one, for example.

So, at least of the level of the data distribution, we can browse existing solutions in order to find out any existing technologies, frameworks, bricks etc.

If we talk about a high-level of abstraction, we can consider technologies like:
- DDS (Distributed Data Service) standard, with some of its implementations (OpenSlice, for example - http://www.prismtech.com/opensplice)
- HLA (IEEE 1516 - http://en.wikipedia.org/wiki/High-level_architecture_(simulation))
- ZeroC ICE (http://www.zeroc.com/)
- DHT protocols and frameworks (http://en.wikipedia.org/wiki/Distributed_hash_table)

Starting with ICE, we enter in fact a new world, the one of low level frameworks providing remote services and, so, data distribution through the services mechanisms.
The collection of frameworks, libraries and standards is huge. We put some below, in an unsorted way: Corba, .NET/WCF, Remote Method Invocation, Apache Thrift, ADAPTIVE Communication Environment (ACE), etc.

Behind this, we could find several underlined standards, like SOAP, HTTP, WebServices... (once again, unsorted and not necessarily similar). It is not the purpose of the current document to list and detail their features, but if we had to do so, we certainly would come

to the conclusion that something like DDS or DHT technics are much more adapted to our aimed multi-platform, scale free, realtime systems.

It is quite obvious that we need to distinguish two different strategies:
- use high level frameworks that provide services similar to the required one (close to the distributed hash table concept);
- develop data distribution services starting from a a low level framework.

During the last three years, we have evaluated both strategies. Our conclusion is that most of the high-level technical bricks available as open source are related to the Web systems engineering. This has only one main disadvantage: the fact that they are not easily embeddable on any kind of embedded target.

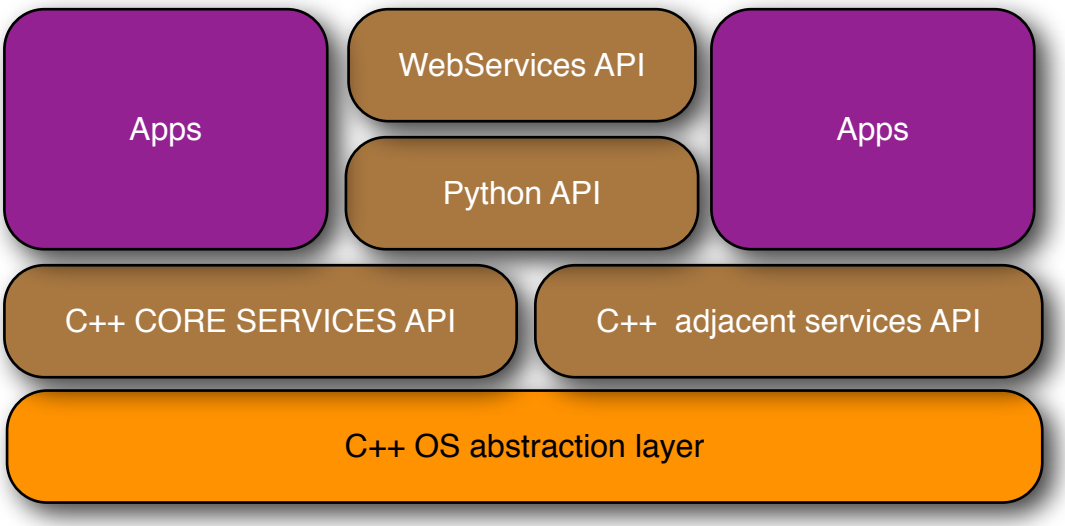
Then, we consider that for the Fractal Systems subject, it would be better to start from low level frameworks.

In terms of programming languages, we face two major streams (if we except Microsoft related technologies, which are not open enough for a large and portable use):
- C++ based frameworks (with all the possible bindings for different languages)
- Java based frameworks.

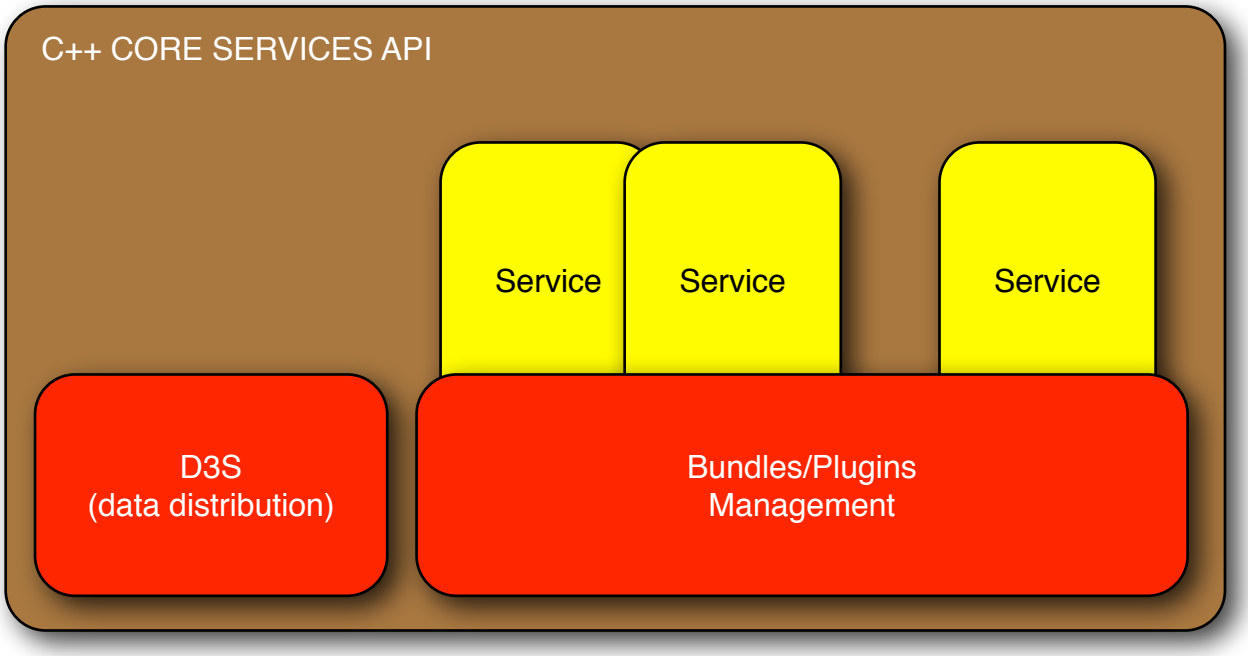
Java is a very interesting choice in terms of portability, development duration, skilled developers availability, code maintainability and long-term obsolescence.
Java has been embedded from years. Some studies (e.g.: JOP [1]) have even explored the possibility to run Java components on FPGA hardware. In spite of all this, we consider, due to our experience, that Java portability is not very easy to manage in what we call a «scale-free» architecture. Then, our approach is definitely based on C++ language and its different bindings.

b. O3S implementation

For our first Fractal System like development (which O3S product - Open System Simulation Solution), we have adopted the following architecture layering:



The CORE SERVICES represents the equivalent of the FSCore and FSCoreAdapter described previously. Below, a more detailed architecture of this particular major component:



Core adjacent services represent all the base framework services not directly related to Fractal System ones. In our case, we are using Poco C++ (http://pocoproject.org/) framework and then, all these services are related to the one provided by the Poco framework.

Thanks to the Poco C++ library, it is possible to address a large set of architectures and platforms. Indeed, Poco has one of the higher levels of portability for C++ source code. It is very fast and efficient, while it provides 90% of the needed low level features. It represents our *C++ OS abstraction layer*.

The current issues of our O3S architecture, are all about porting O3S at the smaller level of a Fractal System.

The first major one, is being able to have the Python API available on embedded targets. If this seems possible for most of the Linux or QNX based targets, it looks currently impossible to do so for firmware based boards, or for FPGA.

Even if this is not one of the major requirements of the Fractal Systems, we have been looking for new solutions for the future implementation.

c. Future implementation

Currently, our most probable choice is, once again, inspired by the Web technologies.

One of the most well known is Node.js.

Of course, we are not the only one thinking at applying this framework to embedded targets. This has been done already for the BeagleBone (http://beagleboard.org/bone) and it is surprisingly interesting.

The reasons are quite obvious, when studying in details the last Web technologies.
Javascript has become the «lingua franca» of the Web. We could compare it to the assembler languages, but also to any high level scripting one. This is a very powerful aspect of Javascript. Whatever we think about it, we can see that its uses are equivalent to both, low level instruction languages and high level one.

For example, taking a look to a site like http://altjs.org/, we can see that, in order to be available on Internet browsers, a lot of programming languages provide a way to compile to... Javascript.
Assembler like, but a lot more. Indeed, if we consider the browser capabilities (with projects like ChromeOS, but also with Native Client project (https://developers.google.com/native-client), it is clear that we can assimilate a browser to an OS. So, compiling to Javascript it is much more than just compiling to an Assembler like language, since we have OS level services directly available.

On the other hand, for projects like Google's V8 (http://code.google.com/p/v8/), we run Javascript almost at the level of performances of a C++ code.

Node.js is based on V8. And V8 is a highly portable Javascript engine. It can be embedded quite easily on boards like the BeagleBone, and, even if the road seems to be long to that, we can imagine to run it on an FPGA hardware.

Issues are numerous:
- compared to Python, Javascript scientific ecosystem does not exist;
- merging Poco with V8 will lead to a potential gas plant, which has to be avoided;
- a choice has to be done between Poco OSP bundle mechanism and Node.js one;
- data types and data encoding in Javascript are not rich enough to cover C++/Python capabilities;
- the capability to run V8 on realtime OS like VxWorks is currently unknown (QNX port seems to be already available).

But future seems to be great for these technologies and providing critical Fractal Systems based on them is a wonderful challenge.

D. Bibliography

[1]

JOP: A Java Optimized Processor for
Embedded Real-Time Systems

DIPL.-ING. MARTIN SCHOBERL, *Technischen Universität Wien Fakultät für Informatik*

[2]

Hazelcast
http://en.wikipedia.org/wiki/Hazelcast
http://www.hazelcast.com/docs.jsp

[3]

JGroups
http://www.jgroups.org/

[4]

PyCPU
http://pycpu.wordpress.com/

[5]

Node.js
http://nodejs.org/

[6]

CAP Theorem
http://en.wikipedia.org/wiki/CAP_theorem