# Anywhere, Anyone Networked !

# The Activator :

# *The way to Adaptive Computing*

## 1. Introduction

We are on the brink of a new era in the evolution of Computer usage: Mobile and/or Distributed Computing as a structural part of the back-office. The ever increasing capabilities of Computer Hardware infrastructures has made it possible to build ever smaller general computing devices. Not only have these devices become smaller but also less expensive, bringing them in the financial reach of ever larger user populations.

Until recently, Mobile Computing was limited to special devices, often tailor made for the projects that these devices were intended for. Being so special, the mainstream IT department generally did not consider these devices as real computers or hardly being a part of the infrastructure: They were looked upon as foreign devices that needed little (data-entry) or no real interaction with the back-office platform. Mobile projects were often considered to be isolated.

Very recently, mobile devices have become more and more popular. More and more people are using these devices, either for their own pleasure or for business. This evolution caused IT departments to shift there attention more towards these devices. The increased power as well as the popularity has caused users to request more and better integration with the corporate data and information structures. Eventually these changes in attitude will lead to the acknowledgement that the mobile platforms, whatever they are used for, will be considered by the company as an integral part of the overall IT infrastructure.

Currently, mobile devices are called "mobile" because the user carries them around. When the user needs to connect the device with the back-office, the communication infrastructure used, regardless if there is a physical wire present or not, is rather inflexible. The layout of the network once the connection is established (= topology) is such that it <u>looks</u> like it uses wires. Indeed, most mobile devices are considered to be satellites of some user controlled 'host' (call it the users' desktop). Whatever physical network the device uses, it will for most (vital) services want to connect to that single host. As a consequence, most services on a mobile device are currently configured to be run on a fixed network layout or STAR-network.

The Internet has provided us with a different kind of network layout. On the Internet ANY device can connect from ANY location and request a communication session to ANY host machine in ANY location. The network is neutral with respect to the logical, application determined, communication that is being established and/or used. Also the calling device refers to the host that it wishes to connect to by specifying a logical network address. This logical address is translated by the network to the physical address of the host. The calling devices will never need to know that physical address. The network topology of the Internet is called a CLOUD-based network layout. There is no difference between the traditional client and server roles : callers and called are peers.

If one could take a snapshot of all communication sessions that are active at any moment in time on the Internet, one would see a rather chaotic layout of these communications. In fact, the Internet itself will (automatically) determine what the physical path of the data from the called (source) to the calling (destination) computer will be. Even stronger, the data will be chopped up in smaller parcels or PACKETS and the network could assign a different route for each of the packets. The network takes responsibility for the disassembly (in the source) and assembly (in the destination) of the transmitted packets. All this activity happens transparently to the application : the flow of data will appear identical on both sides of the communication.

Now that we know that the Internet brings a new view over networking, we can consider its impact on applications and application design. Since applications never use physical addresses of the hosts they wish to connect to, the network does not guarantee what host really actually acts as server. The assurance for finding the identity of the server, will have to be provided by other services on top of the physical network (e.g. Security services). At first glance one could say that it is difficult to imagine situations where the application does care to know the server it connects to. However these situations are in fact not so rare. E-mail for instance. If it were not because of Spamming, no user cares what E-mail server will pick up the message as long as it reaches its destination. Many applications really do benefit from this server-transparency. Techniques like load balancing (multiple parallel servers to serve a larger population of clients), roaming (a different server takes over the connection while the client moves through the network), auto-detection (the client broadcast on the network to detect what printers are available) are all based on network transparency.

## 2. What is Adaptive Computing

Generally speaking, Adaptive Computing refers to application behaviour. It identifies applications that behave in a flexible and ever changing way. The term Adaptive Computing is used for applications that are able to auto-detect the environment they operate in and are able to act upon it. Some Adaptive Applications are able to continue monitoring the environment and if need be, act upon any detected change on the fly.

Although this above description is rather theoretical, Adaptive Computing is not really new (what really is !). Indeed, a good and old example of Adaptive Computing is the way an Appletalk network works. When you plug a Mac into a network, it will start monitoring the conversations (communications) that are occurring and will try to detect the available services (printers, file-servers, ...). When a server falls off-line, the system will detect and automatically reflect the new situation.

Truly Adaptive Applications look at the environment not for what it is but for what the environment can offer with respect to the functionality the application is offering the user. To do this, Adaptive Applications make abstraction of the environment they operate in when designing and implementing functionality. A simple example of such abstraction might be the ODBC-API or SQL. Both techniques allow the application that uses them to be abstract with respect of the underlying physical database (Oracle, Sybase, ...).

Adaptive Software will make many more of such abstractions. Actually it thrives upon them.

## 3. Why do we need Adaptive computing

An application that wants to operate in a truly mobile environment, cannot make assumptions about the environment it operates under in any way: not on the availability (e.g. Connected/Disconnected) or quality (e.g. Bandwidth) of the network nor on the application services that might or might not be available (e.g. Printing). In order to cope with the many circumstances that might occur, an application needs to be very flexible. Classically, writing flexible applications means writing applications that have a lot of configurable options. Because of the myriad of situations most options are only valid in special circumstances. Unfortunately software that tends to be flexible (i.e. Have a lot of configurable options) tend to be difficult with respect to setting them up. Examples are many Unix applications : very powerful and extremely configurable, yet very difficult to set up properly.

Also, this kind of applications tends to be very large and complex. The more environments the application needs to be available under, the more complex the application becomes.

The features list of the application tends to grow longer. Each feature ever so slightly is different from the others requiring also a very subtle design of the generic application parts. This increase in complexity can only be maintained until a certain level is reached. This level corresponds to the moment where the application actually becomes too difficult to maintain and redesign imposes itself. At that point one has to choose between one of these three options :

1. Rewrite and/or redesign to accommodate the new requirements.
   Since the generic parts can no longer easily deliver the required subtlety, they need to be altered fundamentally.
2. Make a 'special' version of the application
   Since most 'common' situations are probably already covered, one could consider the new situation to be too special to be incorporated in the main system.
3. Not support the new environment.

All choices are equally harmful. The first one is costly and does not deliver any guarantee that the rewritten version will fulfil its promises and delivers an application that can not only cope with the new environment (probable), but also have the same performance and quality as the old application had for the old environments before (less probable). The first choice brings RISK.

At first glance, the second choice is cheaper and hence better than the first one. Indeed, one can expect the special version to cost far less than the original version. However the fact that now TWO applications need to be maintained and synchronised, brings with it a cost that could eventually become much higher than a complete rewrite. The second choice can be COSTLY.

The third optional choice, being simpler, has the drawback that the application in effect is dead. It is no longer capable to grow and improve. The third choice is again DECAY.

Worse, the above situation makes one large assumption. We have assumed that one knows and/or controls the new environment. Again this might sound logical, however this new environment might be using new technologies that you or your company might not know about. Even worse, you might not even know the environment at all ! It might be a regional situation or something that occurs in special circumstances.

The complexity together with the uncertainty, makes writing classical flexible applications even more difficult perhaps even impossible.

This is where Adaptive Computing links in. Adaptive Applications are supposed to have implemented their functionality (say Database access) in an abstract way with respect to the environment in which they are executed (say Sybase, Oracle, SqlServer, ...). In other words, the functionality the application provides to the user is correct, REGARDLESS of the real environment in which the application is executed. Therefore adding support for new environments, does not require any application changes at all but will only require code that makes the translation from that environment to the requirements the application has (say a new ODBC-driver for the new database)

It should be clear by now, that Adaptive Applications by excellence are suited to cope with highly volatile networking and operating environments.

## 4. Introducing Component software

We now know that an Adaptive Piece of software handles environmental changes by making abstraction of them.

Now all this remains theoretical. What does an Adaptive Program really look like?

Adaptive applications detect the environment they operate under and will try to find the above introduced 'conversion' software bits. Because of the fact that there are too many (infinite ?) sets of possible needed conversions, the application cannot possibly hope to contain all possibilities in it. Therefore these conversion software bits are external to the application. So the application, after it has observed the environment and determined what it needs, will locate these required extensions and use them. These conversion programs are called Components.

Components share similarities with plug-ins. Plug-ins are designed with the structure of the application that will load them in mind. Therefore plug-ins can only be used by one particular application. Components on the other hand are designed completely independent of the applications that will use them. They behave like self contained and independent little "programs", taking control of everything they need. Components are designed independently of the applications that might or might not use them.
Components could also be looked at as small "drivers" that handle a certain environment resource. E.g. database access, network protocols, printer handling but also business related algorithms might be examples of components.

In fact applications themselves are or could be considered as components loaded by the

OS upon user request.

So knowing that applications use and load components to interact with the environment and knowing that actually applications themselves are in fact components, we can say that Component Based Applications consist of a set of components all interacting with each other. This in contrast with a more classical application design where the application can be identified by the executable that will be started when the application needs to be run.

Which components effectively are a part of the application, depends on the actual environment that operates in any given situation. When the environment changes the application might decide to use other components to interact with the new environment. Therefore, the set of active components can actually change between each consecutive run of that application. Stronger even, the set of components might even change DURING the execution of the application, because the underlying environment might have changed (e.g. Because of roaming).

So the word application becomes a more abstract term. It is no longer the executable that can be run but rather the assembly of components loaded at a certain time to offer the features the user requests based on the environment the request needs to be fulfilled. An application hence denotes more the functionality that is being executed by the users but has no longer a physical representation on the hard disk (EXE file like in Windows).

We have said before that applications make an abstraction of the environment in which they operate, yet still implementing the requested functionality. Using the newly created terminology this translates to : components make an abstraction of the environment they operate in.

How does a Component do this ? Well, while designing its functionality, the component describes the behaviour it expects from the environment. I.e. Each component says : "If I need to do this, I need this behaviour from the environment". E.g. "if you want me to print I must at least have something that **behaves like a** printer". This does not mean that the component needs a REAL printer to print but only something that can behave like one. I.e. As long as the environment can offer something that behaves like a printer, the application will run. You might now say : "What thing behaves like a printer and is not really a printer ?" Well if you select Print->Preview in your favourite word processor you have an example.

So components make an abstraction of the environment by defining behaviours and components hide the environment by implementing behaviours.

## 5. Isn't that the same as Object Oriented Software?

Object Oriented programmers might think : This all sounds a lot like Object Orientation.

True, Component Base platforms and Object Oriented Platforms are cousins of each other. Both implementations hide the resource or environment.
However, the functionality of an Objects (or class) can be determined by figuring out what the class IS. The IS A relationship, made concrete through the class hierarchy.
The API of a leaf class (one that has no more subclasses) is the sum of all parent classes up to the root class.

Component platforms also have instances and these instances also manage the data associated wit one particular usage of that component. However, component instances determine functionality not on what it IS but on what it HAS. This relationship is modelled through the HAS A relationship. Here the size of the API is only determined by the component. Components use (HAVE) other Components that assist the component in doing its job. These internal (or delegated) components are invisible to the caller of the component to which they are internal and hence abstract.

Another big consequence is that since components are designed abstract with respect to the other components, they do not relate to each other at development time (only at runtime), component software is easier to model than Object Oriented designs. The fact that Objects need to belong to a family, requires the developer to make good class designs and think thoroughly about the family tree of these classes. Any error might cause the application to loose any degree of flexibility.

Component based design is more flexible since each component only takes into account only these parts of the environment that are strictly needed for the behaviours that it wants to implement. It does not care who will implement the environmental abstractions it has made nor where the implementation of these abstractions will come from.

## 6. Why are applications that are implemented "in the component way" better ?

Strictly speaking, Component applications in fact do not exist. You only have a bunch of available components (each taking care of some kind of resource - e.g. a database) that are loaded during the runtime of the application, based on the users' wishes. So one can say that an application is running with a subset of all available components loaded in memory. These components interact with each other by the behaviours that they mutually share.

Now suppose that the application needs to be ported to a new environment. We already know that in fact it is not the application that gets ported but new components are created that 'hide' the new environment in such a way that the behaviours needed by the application, are still present. Since the application does not care about the component itself but only about the behaviour they implement, the application will not notice that new components were created and hence a new environment is supported.

Now suppose supporting that new environment is impossible because of a design error in the application. In worst case (i.e. EVERYTHING is wrong) there is only one choice: a rewrite of the application. Fortunately in most cases only some components need to be rewritten. Since the unchanged part of the application is abstract to components (not behaviours!) that part of the application will not notice the change. So because of the redesign of some components the new environment is supported but because of the abstraction the application will still be identical to the "old" application. So instead of **having one new** application we have some enhanced components and still the "old" application

Because it is so easy to modify the design of component based applications, you can use the technique not only to maintain the application but also to do some kind of Rapid Application Development (RAD) on it. Indeed, one could roll out intermediate releases to the customer before the first release is finished and immediately take his remarks into account by accommodating the remarks into this first release. This way the quality of the end-product will be better.

Finally, because of the abstractions one has to make, one can design applications before all technical details are known. Using this technique one can design applications using a stepwise refinement approach.

## 7. What about management?

One could indeed ask oneself : "If there are so many components floating around all being independent of each other, how do I keep track of what is installed or required, where and by whom ?" Indeed, having many components do tend to create some kind of chaos. Component based platforms tend to be 'intelligent'. The Component Runtime Environment (this is the environment that enables the Component Application), will keep track of the versions and the locations and the downloading and installation of new components automatically. When a new version of a certain component becomes available, this environment will detect it and take the necessary steps to upgrade the old to the new version.

Management of this environment becomes extremely flexible. Indeed, the application themselves know what they need.
Now one could say: "Of course applications do know what they need. If something is missing they just fail. True, applications do find or load DLL's or Shared Libraries. If these are missing the application fails and terminates. But Component applications have more information about the feature they need, more than just the name of the library. With this information, the environment can act more flexibly.

So based on these enriched applications' requests and the available components the Component Environment determines which components and what versions of these components are actually needed. This proactive kind of management is far superior to any 'administrative' techniques that are common in classical environments. These administrative techniques generally depend on some kind of release and roll-out management that is maintained and operated by Human Intervention. Developers establish a release, (in the best case) describe that release, and transfer the docs and executables to the roll-out management team. However:

1. Sometimes the release that developers establish contains errors (bugs, installation errors, ...)
2. Developers do not have a lot of time for writing documentation. So if they do write documentation it is source code documentation. However the roll-out team wants to know about application behaviour, resource usage, installation problems, system requirements, error messages and their meaning, dependencies etc. Most of this information is failing
3. Roll-out teams generally are overloaded with distribution request
4. Roll-out teams do not have real knowledge of the application and hence do not really know what will happen when they install the application.

Proactive systems, like Component Based Environments eliminate or should eliminate the roll-out phase completely. In fact, after the release has been made the development team needs to get an OK of the roll-out team. The roll-out team only has to take care of the business necessity of software availability and of the structure of the network The system itself will match the network structure and timing info set up by the management team against the available components and perform all aspects of the installation (distribution, activation and error recovery). So the software-to-be-released, after OK, is basically released into the 'component network' and the dispersed runtimes will pick up the new releases ASAP.

Management of such a network almost becomes a piece of cake. The design of the network generally is a one-time job (similar to the set up of a firewall: define what the organisation wants). The actual distribution of software, happens automatically. In fact, with this system, a "soft" reset could be sent to all systems. A soft reset is similar to a hardware reset. The system restarts from zero and bootstraps itself back into operation. In a Component Environment it means that all software EVER distributed, will be flagged "obsolete" and all devices will resynchronise with the central (but possibly replicated and dispersed) component repositories. This causes ALL components that do not have a newer version to be flagged OK and all components that really are not obsolete to be reloaded. As a net effect of this reset all devices will have the latest version.

Because of this flexibility of software distribution, the perceived quality of the system improves. Users are not prompted with upgrades or yes/no messages and find the application **automagically** upgraded. Fixed bugs find their way to the user's device much faster. So fast, that bugs can be fixed BEFORE the user even detects them.

What happens when the reset arrives when the application is running? Well **our** component environment will have multi version capabilities. This allows running old versions (for as long as they are needed) with sessions based on the newer versions. It could even be possible to roll old versions over the newer version thereby possibly converting the old data to the format needed by the newer version. Imagine seeing that annoying indent visualisation problem of your favourite word processor being fixed, WHILE YOU ARE TYPING DOCUMENTS!!

## 8. What about the embedded market?

Embedded devices can also highly benefit of these techniques. Currently the firmware for these devices is ROM or Flash based. Because of the hardware production cycle, the software for the device is frozen almost 6 months before the device is available for the market. Testing therefore is an important exercise.

Complex software makes testing difficult. There is always another bug to fix. Therefore the decision to release and ship the software is based on time to market, rather than on the length of the bug list. For this reason it is almost inevitable that complex software is being released WITH bugs.

For embedded devices this is really a problem. These devices are designed with heavily reduced human interventions and long-run periods in mind. Remaining bugs do not help to meet these targets.

So, the dilemma is: ship simple software with few (no) bugs or ship more complex software with an increased chance of shipping bugs.

Since the AP Component Environment requires almost no human intervention (remember, devices do not even need to be shut down), the risk at upgrading or altering the software in Component Based devices is importantly reduced. Therefore the remaining bugs by increased software complexity can be fixed more easily. Even before the devices are actually put into use at the customer's site. It can last some time between the release of the firmware of the device and the first usage of that device.

Just imagine how many bugs could be solved in the meantime!!

But there is more. Because of the improved flexibility of the application, using the AP Component Environment, applications can be modified far more easily. This provides more liberty to the developers to fix problems or improve the software but it also increases the expected lifespan of the software itself. Through this **Evolutionary Development** we have created **long life software.**

The harsh competition between producers/vendors of devices, increases the need to add features even to existing/operational devices. With the AP software this can be achieved quick and painless.

With AP Component Based Technology, once the device comes on-line, the firmware of the device can be upgraded to the latest patch level state and the latest set of features, without any human intervention. **"Automagically"**

## 9. So what does the Activator technology of Adaptive Planet offer?

- It offers transparent, network independent distribution and communication facilities
- It runs on many different platforms
- It is implemented in "C", which guarantees extreme performance
- There is lots of experience with local and/or remote database access
- Our technology has proven itself for use with Dynamic User Interfaces
- The platform can sense the available features and auto-detect installed components
- The Activator takes care of, besides the activation of the collaboration between components, 4 basic functions: Error Handling, Bootstrap, Dynamic Memory Checking,

Event reporting

- Every aspect in relationship to distribution, installation and management of operational software, decreases enormously and will even be zero in some circumstances
- "Long life" software because of adaptive behaviour and **evolutionary upgrading at runtime (Software "Hot Swap")**
- Automatic coercion of instances
- **Small footprint (The Activator is only 35K).**

Under conditions the following components are available:

- Database access
- Database reporting (Ad hoc query)
- Scheduling
- Configurability of applications with automated GUI generation
- E-mail interfacing for reporting critical application events
- Components that interface with telephony systems, allowing to build automated and managed call centres
- Data distribution
- Client/Server architecture