AmbientOS Technical Overview

AppInstall Innovation Labs

October 2015

Note AmbientOS is still in a very early stage and there are many unsolved questions, so this document will still change heavily. Some of the open questions are included in this document. They written in this style: *how can we blabla*.

1 What is AmbientOS?

AmbientOS is an ecosystem of applications, services and libraries that adhere to a specific set of carefully chosen concepts and abstractions. These abstractions are designed to make it easy to build applications that are secure, distributed and modular.
Mainline AmbientOS is our distribution of AmbientOS that comes preconfigured with the most essential components you may need and then some.

1.1 API

All API functions are ultimately based on a small set of core functions that enable an application to interact with the world (files, devices, UI, ...). These functions make up the **object store interface**. An application, that relies just on this set of functions, it is highly platform independent by default. This is illustrated by looking at both native and foreign cases:

1.1.1 Native Application

In this scenario, the application is linked to a native-API module, that just issues syscalls for each function call. The actual implementation of the object store runs on the bare metal hardware. Usually, an application can expect the availability of a bunch of core system services (e.g. filesystem, device drivers, ...). Interaction with these services is done through the kernel. The native kernel can make use of an arbitrary set of optimizations to make this interaction more efficient.

1.1.2 Foreign Application

On a foreign system (e.g. Windows, OS X, ...), an application is linked to a fully featured object store implementation that runs in user mode in the same process as the application. An appropriate abstraction layer must be used that contains service wrappers for the host OS functionalities. Since the object store is agnostic to the type of objects it hosts, there are no restrictions on what functionality the host OS can provide.

1.1.3 Differences

Most differences beween running an application natively versus on another OS, are from a usability and capability perspective. *what exactly are the differences? could we use binary translation on another system?*

2 Concepts

Programs expose **Objects**, that have one or multiple **Appearances**, each of which is associated with an **Interface**.

2.1 Objects

Objects are the only way for programs to cooperate. If a program wants to expose a resource, it does so by publishing an object. If another program wants to use an object, it invokes methods on a suitable object. Examples of objects include: files, folders, monitors, speakers, sensors, actuators and much more. Even programs are objects.

2.2 Interfaces

Interfaces define what operations are possible on an object. An interface definition encompasses a globally unique name, a set of methods and their signatures and a set of attributes.
An object can (and often does) implement multiple interfaces. For instance, an external display may implement both the `AmbientOS.Graphics.Screen` and `AmbientOS.Audio.Speaker` interfaces.
For programs to do anything useful, it is essential that they can use most of the objects out of the box. That's why Mainline AmbientOS predefines interfaces for most common purposes. These interfaces all belong to the namespace `AmbientOS` and all Mainline AmbientOS components are based solely these interfaces.
For a complete list of Mainline AmbientOS interface definitions, see *... we need to generate interface documentation from XML description*. For instructions on how to design your own interfaces, see section 4 on page 4.

2.3 Interface Implementations & Object References

It is important to distinguish between a reference to an object and an implementation of an interface. The interface implementation lives within one single program. In general, direct pointers to implementations should be used with caution. A reference to an object *unfinished* acts as a proxy that forwards method calls either to a local implementation or to a messaging system. While an implementation can implement multiple interfaces at once, a reference is always associated with a particular interface. References have multiple advantages over direct pointers to implementations:

- Whether the implementation of a reference resides in the same process, in a different process, or on an entirely different machine is irrelevant to the client.
- The implementation can change or move without the client having to care. For instance, a file could be renamed, or even moved to a different file system while it is being used by other programs.
- A reference can consist of multiple paths to the same object. If one path breaks down it can seamlessly transition to another path. Similarily, it can use multiple paths concurrently to increase bandwidth, or it can select the best path depending on an appropriate metric, such as cost, latency or others.
- Through reference counting, the implementation always knows when it's no longer used.

To prevent unneccessary messaging overhead due to reference counting, references are hierarchical. Each implementation maintains a root reference for each of its interfaces. If other programs want to use the object, a new reference to this root reference is created and the reference count of the root reference is incremented. When the reference count of the root reference reaches 0, the implementation knows that it's no longer used and it can free the resources associated with that interface.

2.4 Reference Counting

Reference counting consists of two operations: `Retain`, which increments the reference count and `Release`, which decrements the reference count. It can be hard to know the right places to call these functions, so we need to set up a set of rules (here, object denotes any reference counted object):

- Functions that return an object: The function must retain the object before returning it. The object must release it when it's no longer used.
- Functions that take an object as a parameter: The caller must retain the object before the call and release it afterwards.
- Classes or structures that hold an object as a field: The object must be retained before being assigned to the field. When another object is assigned to the field, the old object must be released. When the structure is no longer used, all reference counted fields must be released.

2.5 Programs

Applications, services and drivers are all equivalent in AmbientOS. Let's call them **programs**. The way programs are executed depends on the system setup. On a normal PC, programs run in the form of processes in user mode. On an embedded system or on processors where there is no memory protection, programs run alongside the kernel. Conceptually, programs may also run on FPGAs or anything else that can do processing. Installing a program is equivalent to publishing an object with the "AmbientOS.Framework.Application" interface. Since programs can only expose objects while they are running, a program usually does not expose itself, but instead registers itself in an app registry. The app registry can be found by searching an object with the "AmbientOS.Framework.ApplicationRegistry" interface. *how to select the best app registry?* The default app registry is tightly coupled with the boot process of the local kernel, e.g. it may store registrations on disk and on startup publish an "AmbientOS.Framework.Application" object for every program it knows or finds.

2.6 Peer

A peer is a single self-contained component of the AmbientOS network. Each peer has it's own object store and can host multiple programs. On a native AmbientOS system, each processor in the machine is a peer. On other host OSses, a process is a peer. Connectivity to other peers must be provided by the programs running on the peer. This can include TCP/IP, bluetooth, shared memory or any other method.

2.7 Domain

A set of cooperating peers can make up a domain. A peer can be part of multiple domains. A few examples:

Local peer This is a built-in domain that cannot be altered.

Local machine Contains all of the peers (i.e. processors) on the local machine.

My devices Contains all of the devices that are (co-)owned by the user.

Family Contains all of the devices owned by the user's family.

Home Contains the devices at home (e.g. lights, sensors, heating, ...).

Company Contains all of the devices that are part of the user's company.

Cloud Contains services provided by a cloud provider.

Mainline AmbientOS will include tools that make it easy to manage domains.

2.8 Object Store

The object store is the central manager of all objects. It supports two main operations:

Publish Publishes an object in all domains that the peer is part of.

Query Lets programs search for objects of the specified appearance. A query is active until it is terminated, i.e. a callback is invoked whenever a new object was published or disappeared. A query is always associated with a domain. *what kind of queries do we want to allow? only search by object type? or by attributes? how about regex? in the end we can't cover every use case, so the client will need a filter anyway, but still we don't want to flood it with objects*

3 Design Principles

include the design principles that we follow (mainly concerned with user experience)

4 Interface Design Guidlines

For a lot of common use cases, there's already a predefined interface in the `aos` namespace. However, you're completely free to define your own interface to enable interoperability with others. To make sure that others get the most out of your interface, your should consider a few things.

- Don't only think about your own use case. Think about what functionalities others may want to provide if they use your interface.
- Make the interface general, rather than easy to use. If you want your interface to be easy to use, provide a library.
- Keep in mind that some implementations may lack some of the functionality of the interface and specficy the behavior in these cases.
- Find the right set of security priviledges. Think about what subset of actions the user may want to allow on the objects.
- Be clear about the semantics in the documentation.
- Use it yourself. This is the best method to find flaws in the design.

If your interface is good and seems useful, you can propose it to us and we may include it with Mainline AmbientOS.

5 AmbientOS.C# Framework

To make AmbientOS development easy, we provide a complete framework that contains a full implementation of the object store, all of the Mainline AmbientOS interfaces, a bunch of useful extension functions to these interfaces and a bunch of other utility function.
In addition to that, we provide abstraction layers for Windows (desktop, console services and universal apps), iOS and Android. These libraries are implemented in .NET, so they are usable by any .NET language.

The framework consists of the following components:

- `AmbientOS.Core` contains all the interfaces, extension functions and some utility functions and classes.
- `AmbientOS.Native` contains the P/Invoke stubs for the native AmbientOS kernel.
- `AmbientOS.Foreign` contains wrapper classes around parts of the .NET Framework that are available on all platforms but don't conform to AmbientOS conventions (e.g. file access, networking, ...).
- `AmbientOS.Foreign.Windows` contains wrapper classes and P/Invoke stubs specific to Windows.
- `AmbientOS.Foreign.Mac` contains wrapper classes and P/Invoke stubs specific to OS X and iOS.
- `AmbientOS.Foreign.Android` contains wrapper classes and P/Invoke stubs specific to Android.

Beware that the names of these assemblies do not reflect the namespaces that they cover.