ACCESS Application Framework

Technical Overview

version 0.9.1, Dec 22 2006

1. What you can do with the Application Framework source release

The source code accompanying this white paper contains the "application framework" source code making up part of the Access Linux Platform. We call this release of the application framework "Hiker".

The source code is useful to developers who wish

- to learn what features the framework offers,
- to demonstrate its use in simple console-oriented development scenarios, and
- to evaluate the services the framework provides.

This is a preview release. We intend to make another release before Spring 2007.

2. Who this white paper is intended for

This white paper is intended for software developers, managers, and others interested in learning about the Access Linux Platform "application framework". The paper provides a detailed look at key pieces of the framework, and shows some of the library calls that are available to those developing software applications for the platform.

You need to have experience with Unix or Linux, and C programming to get the most from this paper.

3. What is an Application Framework?

The term *application framework* means different things to different people. To some, it is the GUI toolkit that is used. To others, it is the window manager. Still others see it as the conventions for starting and running a process (e.g. on Linux, a C program should have an entry point called "main" that takes a couple of parameters and returns an integer). When we use the term "application framework" in this paper, we mean ***the set of libraries and utilities that support and control applications running on the platform.***

Why are additional libraries needed to control applications? Why not just use the same conventions as on a PC: choose programs off a start menu and explicitly end an application by choosing the "exit" menu item or closing its window?

The reason is the different "use model" on handheld devices. PCs have large screens that can accommodate many windows. Based on experience refining PalmOS, we believe that there should be only one active window on a handheld device. As another example, when the user starts a new application, the old handheld application should automatically save its work and exit.

As well as the task of managing the lifecycle of programs (launching, running, stopping), the application framework has three more tasks. Its second task is to help with

distributing and installing applications. The conventions are simple: an application and all supporting files (images, data, localizations, etc) are rolled up into a single file (like a zip file), known as a "bundle". Bundles are convenient for users and third party developers. They allow software to be passed around and downloaded as a single file.

The third task of the application framework is to support common utility operations for applications. These are services like communication between applications, keeping track of which applications handle which kind of data content, and dealing with unscheduled incoming data like phone calls or instant messages.

The final task of the application framework is to implement a secure environment for software. That means an environment which resists attempts by one application to interfere with another (this hardening is called "application sandboxing"). The secure environment also supports security policies for permission-based access to resources. E.g. part of a policy might be "only applications from the vendor are allowed network access". The security policy is implemented using a Linux security module.

Summary: responsibilities of an “application framework”

- managing the lifecycle of applications
- help with distributing and installing applications
- support common utility operations, like IPC
- support a secure environment (avoid software viruses that spread by phone)

4. Getting into the Code

The application framework performs a similar task for bundles, that a browser does for Java applets. It provides a context in which they run, and it is responsible for starting and stopping them when needed.

To give you a sense of the effort needed to understand it, learning the Hiker framework is much simpler than learning how to use CORBA or DBUS. It takes about the same amount of effort as a Java programmer learning how to write a Java servlet or applet.

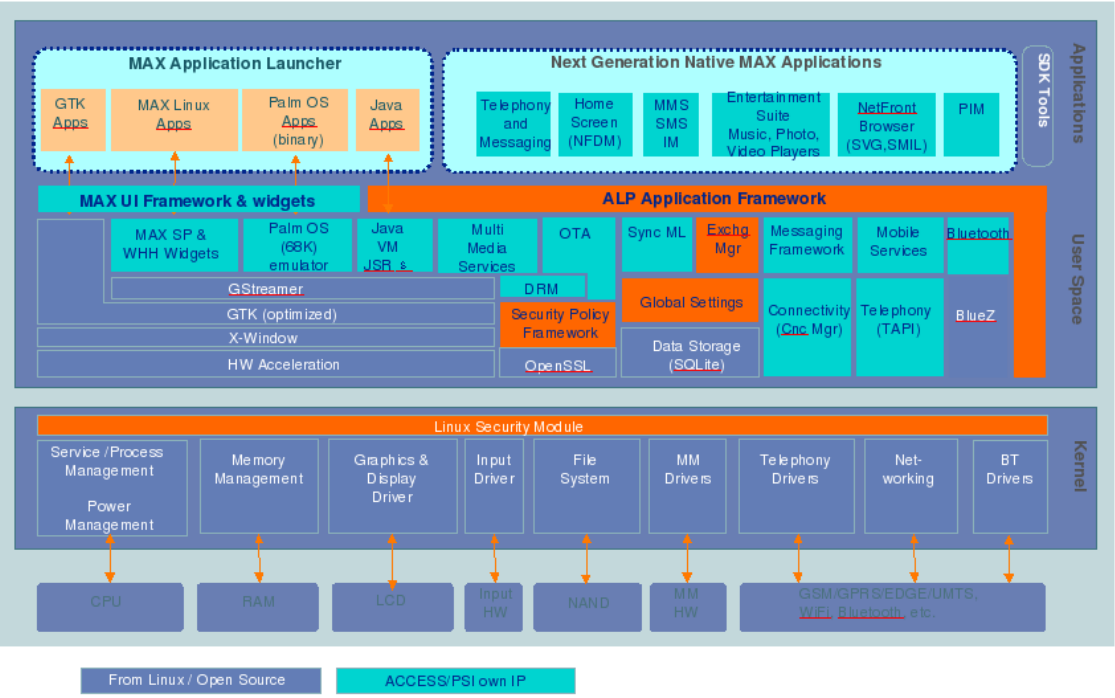
As you read this paper, you will see two overlapping levels on which you can engage with the application framework:

- Understanding and evaluating what the framework offers
- Writing applications which will run in the framework

This paper is more concerned with helping you understand and evaluate the framework, than with teaching how to write applications for it.

4.1 What is included as Open Source?

The Access Linux Platform is made up of many components, from the Linux kernel up, as seen in figure 1 below. Some of these are open source, some not.



The parts of figure 1 colored orange have been released as open-source. The press release is at http://www.access-company.com/news/press/Current/122106_hiker.html.

The source code in a tarball is at http://www.accesscompany.com/about/opensource/download.html

That includes about one dozen components, described at length in section 5 below. There are many additional components making up the complete Access Linux Platform. The complete platform includes extensive GUI, telephony, and other code which is not part of the Hiker open source release.

The Application Framework is almost completely independent of hardware and GUI. It can reasonably be used as the basic environment for any kind of handheld computer running Linux. The application framework includes a number of unit tests that demonstrate use of the Hiker services.

4.2 Source File Organization

In the first release of the framework, there are about 100K lines of application framework code, located in about 500 source files.

When you unpack the tar file, it will create the directory hiker/main which contains the following subdirectories and contents:

directory name	contents

libraries	client side code to access a server
servers	server code to provide a service
whitepaper	some documentation
samples	some example code for you to review and try
doc	The HTML pages describing the public APIs
utils	support for the unit tests, but the interesting part – the unit tests themselves – are in subdirectories called “tests” several levels under “servers” and “libraries”. The unit tests for a component will give you a good idea of how to use the API. The unit tests are written in C++ because the cppunit test tool needs that.

There are other files and directories, but these are the places to start your evaluation.

Most of the services in the framework are implemented by daemons running in the client/server model, communicating over IPC. To make things easier for application programmers, we wrote client side libraries that can be linked with user code, and which handle the IPC to their corresponding server. With this approach, a programmer can access the framework services using (simple) local procedure calls instead of (more complex) IPCs.

4.3 A word about Doxygen

Doxygen is an open source documentation system for C++, C, Java, and other programming languages. It can generate browsable documentation (in HTML) and/or an off-line reference manual (in PDF) from a set of source and header files that contain specially-formatted comments. The documentation is extracted directly from the source, which makes it easier to keep the documentation consistent with the source code.

The API for these libraries has been generated as a set of HTML files using the doxygen open source utility. The public APIs in the source code is marked up with comments in a way that can later be extracted and formatted into a web page. The generated HTML files can be found in directory **doc**. If you cannot find a doxygen page for some API, it means it is not part of the public API.

4.4 Building the code

This release was built on Debian Linux, Ubuntu 5.10 distro, using gcc 4.0.3. The use of the autoconf gnu build system means that you can likely build it on other Linux distros, and possibly on some Unix systems. But if you encounter problems, fall back to the Ubuntu 5.10 distro.

Please refer to the readme file in the root directory of the tar file for instructions on building from source.

The SDK and PDK

ACCESS intends to release a Software Development Kit (SDK) for application developers. The SDK will contain all the libraries, header files, GUI tools (like Eclipse) which help third party software developers write applications. The SDK is not available at this time.

ACCESS also intends to release a Product Development Kit (PDK) for telcos, product licensees, carriers etc. The PDK will allow those organizations to modify and customize the Hiker software. The PDK is not available at this time.

In the absence of the PDK and SDK, **you may need to install several open source components, to build the Hiker code successfully.** The usual way to find out what you need to install is to try the build, and if it fails, see what is missing and install it.

On Ubuntu 5.10, these open source components had to be installed before this release would build: cppunit (1.10.2 or later), libgnet-dev, libdbus-1-dev, libgtk, sqlite3, doxygen. These were installed from debian archive servers, using the synaptic install utility. (The "-dev" suffix indicates it is the development version with header files). If you get compiler warnings about missing a symbol that belongs in one of these libraries, check whether you have an old or obsolete version of the open source library in question.

The README file at the top level on the tar tape has information about building the source. You need to untar and build as root, for a couple of reasons relating to minimizing changes to the Framework while releasing to open source. Building and running Hiker will install some code in the directory /opt/alp (the unit tests expect this hard coded path, for example).

5. Key Components in the Framework

As outlined above, there are four main roles for the application framework,

- to package and install applications
- to start and stop applications automatically
- to support common utility operations for applications
- to support security and "application sandboxing" for application software

Architecturally, the framework is divided into a number of libraries and daemons that operate fairly independently. These libraries are called "managers" or "servers". Each manager works to support one or more of the roles listed above.

installing applications:	Bundle Manager
starting and stopping applications:	application server, native launchpad daemon
common utilities:	global settings, alarm manager, notification manager, attention manager, exchange manager.
security:	described in a later paper

In diagram form, the managers have this appearance. Global settings is a configuration-and-settings utility used by all the other components. It uses SQLite.

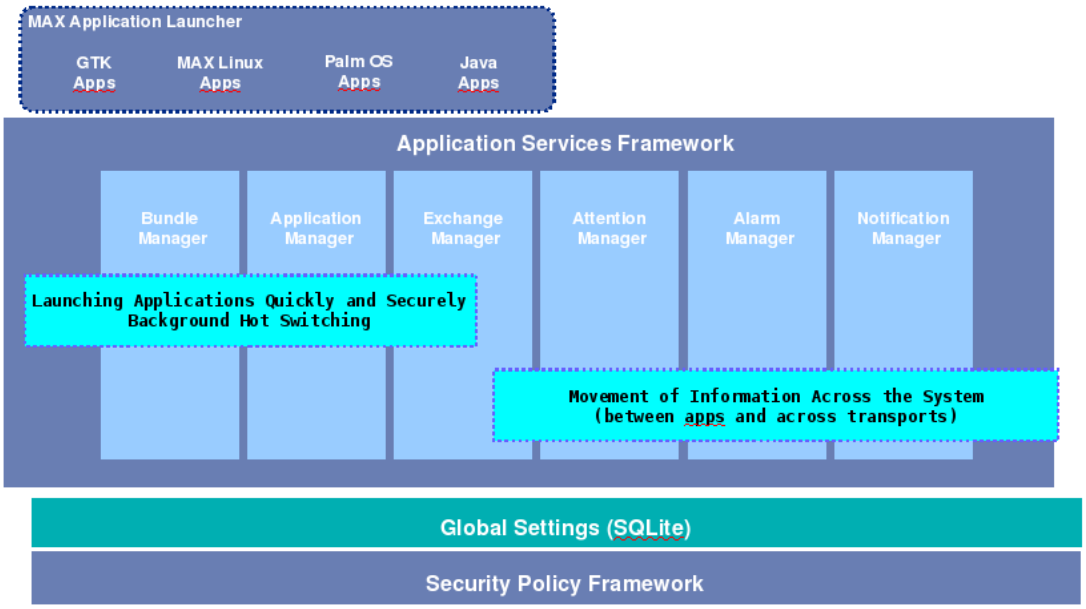


Figure 2: the Application Framework

The following sections review these components. The components are broadly similar to those in previous releases of PalmOS, but modernized and aligned with Linux. The components can be summarized as:

- Application Manager—handles application "lifetime" and execution
- Bundle Manager—defines a format for "packaging" applications, allowing standard access by the Application Manager
- Notification Manager—implements a general mechanism for sending software "alerts" between one running process and another
- Alarm Manager—provides a service of time-based software alerts

- Attention Manager— the foundation for communicating system events to the user (independent of any specific UI)
- ALP IPC—a high-level API for efficient interprocess communication which can be based on any of several POSIX IPC mechanisms
- Security Framework—a combination of a Security Policy Manager and a Linux Security Module to control access to system facilities and resources
- Exchange Manager—a central facility to allow the sharing and use of structured information (e.g. appointments, tasks, songs, etc.) between different devices and different applications on a device
- Global Settings—a common API and store for user preferences, system settings, configuration information, etc.

6. Installing Applications

ALP applications are packaged in a format called a *bundle*. A bundle is a collection of files stored in a cramfs disk image (similar to a .iso or a Mac ™ .dmg file), and usually cryptographically signed. A bundle always contains a file called "Manifest.xml". That file contains meta information about the rest of the bundle: what it is, default settings, preferences, etc. All applications are packaged as bundles, but things that are not applications (like data) can also be packaged as bundles. By wrapping an entire application as a bundle, users and programs can deal with arbitrarily-complicated things as a single file. Bundles have the extension ".bar" for "bundle archive".

An application is linked as a dynamic library (by using the "--shared" option to gcc), with the entry point "alp_main()" instead of the conventional main(). You can see a sample "hello world" application in directory hiker/main/samples. You can see the linker options used in the Makefile in the same directory.

6.1 Bundle Manager

The Hiker component that processes bundles is Bundle Manager. Bundle Manager is responsible for controlling how applications, and supplemental data for applications (libraries, resources, etc.), are loaded onto an ALP system, manipulated, transmitted off to other systems, and removed.

In a real system, new applications will appear on the device through the exchange manager, or through the insertion of a memory card. There are no corresponding events in our sample code, so instead in our demo, we inform the Bundle Manager of a new app by a call to `alp_bundle_register()`.

The Bundle Manager is the sole way in which all third-party applications are distributed and loaded onto a device. Applications are allowed to read the files in their bundles directly, via the POSIX file i/o routines. Apart from that, the Bundle Manager server is the only software that can access the bundle folders on the internal filesystem. Users have to interact with Bundle Manager to install or remove software, and that software must be in bundle form.

Although binary executables could be loaded onto a device through other means (say, an FTP client dumping a file to the /tmp folder), the launcher and other system components would not provide any UI to launch that executable, nor provide any means to execute it with other than the most restricted permissions.

Bundle Manager also supports:

- Finding bundles by name
- Finding bundles by a variety of other characteristics
- Obtaining information about a bundle
- Moving bundles between stores
- Finding what bundle code is running from
- Opening a bundle for direct access
- Retrieving files from a bundle
- Manipulating flattened bundle images, with streaming APIs
- URL convenience routines which allow a consistent URL format to reference any bundle and resource/library on the system. These are names in reverse DNS form, like `bar:com.access.apps.phone`
- Application helper routines to retrieve localized icons and names for an app in a bundle

There is a GUI component called the "Launcher" that makes extensive use of Bundle Manager to retrieve information about available applications, and display it in a GUI. Since it is a GUI component, and Hiker is GUI-neutral, Launcher is not part of this release. Bundle Manager knows about all bundle types, permitting a single launcher to handle native (ARM), Java, and M68K applications. The Hiker release only includes code for native ARM applications.

There is a brief perl script called "bartender" that packs/unpacks between folders and bar files. You will find this in the directory `/opt/alp/bin`

7. Starting and Stopping Applications

From experience with PalmOS we conclude that handheld devices work best when apps don't try to share the limited screen space. Only one app will use the GUI at a time. Furthermore, you almost never exit an application explicitly. Typically, you start another application (by pressing the home key or the launch button, etc). The Application Server component then ends the current application for you, and starts the new application in its place.

There are no application buttons in our sample code, so instead we explicitly ask the application server to tell the native process launcher to load the shared object that is the application and then to jump to the app entry point in the shared object, the function alp_main(). The routine that we call to achieve this is listed in the next section.

7.1 Application Server

The application server or "appserver" is a Linux daemon that starts when the system boots. From the background, it provides these services:

- provides a way to start applications, and control their lifetime
- prevents more than one instance of the same app from running
- may also provide default behavior for system events (such as handling dedicated keys, or clamshell open/close).

You can review the public API for the appserver at doc/html/appserverclient_8c.html. You will see routines like

```
alp_status_t alp_app_launch

      (const char *pkgID, int argc, char *const argv[], pid_t *outPID)
```

That routine tells the application server to tell the native process launcher to start the the stated application.

Note: bundles used to be called "packages" and some of the header file names still reflect that.

7.2 NLPD

We want applications to start as quickly as possible, and we structured the launch code in an unusual way to achieve this.

The largest part of application start up time is consumed in the run-time linker. The run-time linker gets the shared libraries ready to run and initializes all of the global variables. Most application code (on handheld devices) uses a small number of the same shared libraries. We reduce start-up time by having a process which is pre-launched and pre-initialized with common libraries already loaded into the address space. This application is called "nlpd" for Native Launch Pad Daemon.

When nlpd receives the "launch" order from the application server, it forks a copy of itself. That copy then uses dlopen() to load the application shared object and invoke alp_main(). This is significantly faster than the conventional alternative of vfork() followed by an exec().

This performance optimization requires an application to be a shared library, so it can be loaded into the address space using dlopen. You can see the code in file servers/NLPD/Native_Process_Launchpad.c. This is the line that causes the dlopen:

```
    g_closure_invoke(ptr, &return_value, 3, params, NULL);
```

g_closure is a routine in glib that abstracts dlopen.

Nlpd sets the current working directory to the directory where the bundle is stored. The application can thus retrieve files, e.g. icons, stored in its bundle. Nlpd also makes some security- and identity-related settings. Because it is the general template for starting a new application process, nlpd is also called the "cookie cutter".

7.3 The Lifecycle of an Application

Application Startup

The lifecycle begins when the nlpd calls the main routine in an application library. An argv argument tells the application if it is to be the main window, or run in the background, or whether it was started to handle something specific. You can see the argv strings in the source at `/hiker/main/dist/include/hiker/appmgr.h` or in the doxygen documentation (once you build it). Here are some of the strings, and their effects on the application.

```
#define     ALP_APP_PRIMARY   "--alp-primary"
    // The application becomes the new "primary" UI app.
    // Shut down the current app and run this app instead

#define     ALP_APP_BACKGROUNDED   "--alp-backgrounded"
    // The application is no longer the primary UI app,
    // but let it continue in the background with no GUI visible

#define     ALP_APP_NOTIFY   "--alp-notify"
    // The application is being launched or relaunched
    // to handle a notification.

#define     ALP_APP_EXCHANGE   "--alp-exchange"
    // The application is being launched or relaunched
    // to run an exchange handler.

#define     ALP_APP_ALARM   "--alp-alarm"
    // The application is being launched because one

    // of its alarms went off.
```

The most common launch argument is ALP_APP_PRIMARY. An application is typically launched with ALP_APP_PRIMARY, and will continue execution until a new primary application is launched. For the most simple applications, nothing beyond basic GTK (or other GUI toolkit) infrastructure may be needed. More complex applications, however, need some extra work beyond the basic GTK code, and this is where the other launch arguments come into play.

Application Execution

Applications generally run like any GUI application. They use a toolkit like GTK to display some UI, and run a window main loop until it is time for them to quit.

Application Re-execution

The major departure from the usual desktop environment is that we only permit running a single instance of any given application. If a running application is launched again, it keeps running but the new launch arguments are delivered to it and the application is expected to update its state accordingly.

Applications receive new launch arguments through a callback that is termed a "relaunch handler", which is put in place by calling alp_app_set_relaunch_handler(). This handler will then be called later by the appserver client-side library in response to a launch request for the already-running application. It gives the application the opportunity to examine the new launch arguments, and update its state. This might mean handling an exchange request, showing or hiding UI, etc.

In other words, if the application is already running, the framework can tell it of a new task that must be done. And if an application is not running, the framework can start the application to ask it to handle a particular task. In this case, the application generally runs in the background with no GUI displayed.

Applications which might be re-launched include those that support/require both primary and background execution, register for system notifications, or register with Exchange Manager to handle some particular content types.

It's worth noting that unless the application goes out of its way to use Framework-specific facilities (by specifying an ALP_APP_BACKGROUND_* property in its Manifest.xml file, registering for notifications or alarms, registering with the ExchangeMgr, etc), then it will probably never be launched while it is already running, and thus there is no need for it to register a relaunch handler. This provides an easier transition for developers or software porters familiar with the GUI toolkit but not Hiker.

The app server will shut down and restart an app, if it gets a launch request for an app that is already running but which does not have a relaunch handler.

Application Shutdown

We maintain the traditional Palm OS model of having the system tell the application when to exit. While an application may exit at any time of its own accord, it should also be sure to always honor an exit request from the appserver.

Applications which need to perform cleanup before exiting will register callback handlers to receive the exit request, and should respond by performing any necessary cleanup, and returning from main(). The application may add (and remove) multiple callbacks in order to associate different types of cleanup with the current program state (for example, if the application displays a dialog, it can also register an exit handler to dismiss that dialog). Exit handlers are always executed in order, starting with the most recently added.

If the application does not register any exit handlers, then the appserver client-side library will raise a SIGTERM signal instead. This provides a familiar (to Linux developers) mechanism for exit notification. For most applications, the programmer doesn't need to write a SIGTERM handler and can just let Linux end the process.

If the application does not cease execution within a short timeframe (a few seconds), then the application server will send a SIGKILL, and a warning dialog may be displayed to the user (as much to encourage developers to handle the notification as to warn users). There will not be an API available to applications for resetting this timer. Applications with a genuine need for a lengthy exit procedure should instead specify

ALP_APP_BACKGROUND_REQUIRED in their manifest file, and register a relaunch handler. When the app server relaunches the app in the background, it can run its lengthy exit procedure and return from main() to terminate.

8. Common Utilities

The previous sections described how applications are registered and run. This section describes some common utilities in the framework.

Notification manager

> Notification Manager provides a mechanism for sending software notifications of unsolicited system events to applications. Applications can register to receive particular types of notifications that they are interested in. The Notification Manager can deliver notifications not only to currently running applications, but also to applications that are registered to receive them but are not currently running.

Notifications are general system or user asynchronous events, such as: application installed or uninstalled, storage card inserted or removed, VFS mount/umount, incoming phone call, clam shell opened or closed, time changed, locale changed, battery level dropping to low power, or device going to sleep or waking up.

The Notification Manager has a client/server architecture. The Notification Manger client library uses the ALP IPC framework to communicate to the Notification Manager server.

Notification Manager server. The Notification Manager server is a persistent thread in a daemon process. It keeps track of notification registrations, and broadcasts notifications to registered clients. The Notification Manager server also communicates with the Bundle Manager and the Application Server.

Notification Manager client library. Client processes call APIs in the Notification Manager client library to

1. register to receive notifications,
2. unregistered previously registered notifications,
3. signal the completion of a notification, and
4. broadcast notifications.

What the Notification Manager is not.

The Notification Manager should not be used for any old thing that one application thinks others may like to know about, such as the results of a “find” search. The Notification Manager facilitates the sending and receiving of notifications but it does not itself broadcast notifications. Individual component owners are responsible for broadcasting their own notifications.

8.2 Global Settings

This is a component that can store settings for applications. It can store preferences like font size or themes. It can also retain a hierarchy of related settings as used in the OMA standard for device management of phones. Global Settings uses the recently open-sourced libsqlfs project layered on SQLite to store its data.

The source code can be seen at:
`hiker/libraries/global_settings`.

The settings keys may form a hierarchy like a directory tree, with each key comparable to a file or directory in a file system. E.g. a key may be "applications/datemanager/fontsize" with a value of “8”. Indeed, Global Settings can be implemented on top of a file system. Due to file system limitations in the mobile environment we implement it differently (on top of a single sqlite database file). But conceptually Global Settings tree follows the logic structure of an POSIX file system; each key has a value and meta data, such as a user id, a group id, and access permissions.

The Global Settings service has two parts: the daemon (server) and the client library. The client library is only there to make it trivial for the clients to talk to the server. The client library is linked with the client application, and handles the IPC to the server.

The client library and the daemon communicate via ALP IPC. The daemon does all I/O for the data; it uses the SQLite library and does the key content reads and writes on an SQLite database. The tree hierarchy of keys is implemented using relational tables through SQL. SQLite provides no effective access control, so the daemon uses Unix file access control on the database file to exclude the under-privileged. The daemon also keeps track of the users and groups that are allowed to access certain keys, and enforces access control. The SQLite database files are only readable and writable by the daemon process.

We started out hoping to use Gconf for global settings. To better support the OMA-DM we dropped the GConf design, as it does not provide security restrictions to data access.

Data Model

Summary: global settings stores key-and-value pairs. A key looks like a pathname, e.g. /a/b/c/d. There can be a hierarchy of related keys, e.g. /a/b/c/d1 and /a/b/c/d2. This is

very useful for supporting OMA device management. Not only do keys look like pathnames, they resemble them in other ways too, such as access permissions permitted to programs using the global settings library.
Here is some terminology:

1. We expose key/value pairs. Values are also called the "contents" of a key and can be arbitrary data of arbitrary size. If you are storing file size chunks of data, it might be better to store it in a file, and store the pathname in the global settings database.
2. Key descriptions (in multiple languages) are supported by conventions (see below)
3. We support access permissions to keys that are user, group, and other readable/writable (or not); the user and group identities are based on these of the system and are granted by the system. The Global Settings service does not give special meanings to any specific group or user id.

Null Values

When a new key is created, it can have no values, or just the null value. This value has the type "INVALID" and a size of zero. This is useful for creating a key and setting the value to something later. This is also required for creating a directory, because you need to create the key for the directory entry first, as described below.

Value data types

Global Settings supports the following data types for key values:

- int (integers)
- float (double in C)
- string (zero-terminated string in UTF8)
- bool (boolean values TRUE and FALSE)
- blob (an binary buffer with a length)
- dir (used to represent directories in the key space. It has no value itself but contains child keys)
- list (a list of int, float, string or bools)

There are convenient APIs for reading and writing all these data types, except lists.

These data types have "canonical type names" which are the names in the above list. For example, "int" is the official name and "integer" is not a valid type name. The canonical type names are used in API names, C constants and string representations of the types in the Global Settings database and the fields in the initial settings XML files.

List value representation

There are no APIs provided for manipulating list values. The application programmer must do list housekeeping for him or herself. Global Settings list values are represented in memory using the glib data structure GList, with each element represented by a GList node whose "data" member points to an AlpGlobalSettingsValue.

Note Global Settings allows lists of lists and other constructs using lists. Global Settings also provides the mechanisms for serializing and deserializing such data structures, so they may be stored as key values in the Global Settings database.

Default values

We use a simple mechanism for supporting default values. For a possible key S, we hold that it has a default value if another key named S.default exists. If the key S does not exist, a “get key” operation on S will return the content of S.default.

Key Conventions

Key names should follow a standard convention to allow grouping of similar attributes under the same part of the key hierarchy. We follow the GConf conventions as closely as possible for application preferences and system settings, with consideration for device-specific standards like /dm for OMA-DM. See http://developer.gnome.org/doc/API/gconf/conventions.html

Some typical keys are:

> /dm for OMA DM
> /capabilities for items like "is java installed", "is ghost installed" etc
> /packages/com.access.apps.myapp for preferences belonging to "myapp"

Data Security

The Global Settings service is appropriate for data where access controlled by file mode is adequate. The data in the Global Settings are protected by POSIX file permissions, but are not encrypted or Digital Restriction Managed.

Change Notification

Applications can ask to be informed of changes in settings (their own or others). To do this, the application will register a callback with the system's Notification Manager. The Global Settings daemon sends Notification Manager a string representing each key/value

that it changes. Notification Manager will then notify each app that has registered to be told about that key changing. Notification Manager will notify each app that registered for that key, or a prefix of that key. E.g. an app that registered for "oma/apps" would be notified when any of the following keys changed: "oma/apps/calendar/fontsize", "oma/apps/date/background-image", or "oma/apps".

Following the conventions of the Notification Manager, each notification of a change in key/value has the "notification type" (or "notify type") of a string of the form (literally) "/alp/globalsettings/keychange/" to which is appended the key string. For example, a change of the key "oma/apps/calendar/fontsize" will invoke an alp_notify_mgr_broadcast() call with the notification type "/alp/globalsettings/keychange/oma/apps/calendar/fontsize"

The Notification Manager is responsible for monitoring the key changes in a directory tree. It implements this by checking if a changed key has, as a prefix, a substring which matches the key of a key subspace being monitored for changes by some application.

For example, the previous key change example will invoke change notifications for applications which want to know of key changes in the /oma/apps/calendar/ key (directory). The Notification Manager is the component which checks this and invokes the notification callback in client applications. 

8.3 Alarm manager

> The Alarm Manager provides a mechanism to notify applications of real time alarm (i.e. time based) events. Both currently running and non-running applications can receive events from the Alarm Manager. The Alarm Manager has no GUI and does not control presentation to the user – the action taken by an application in response to an alarm is defined by the application.

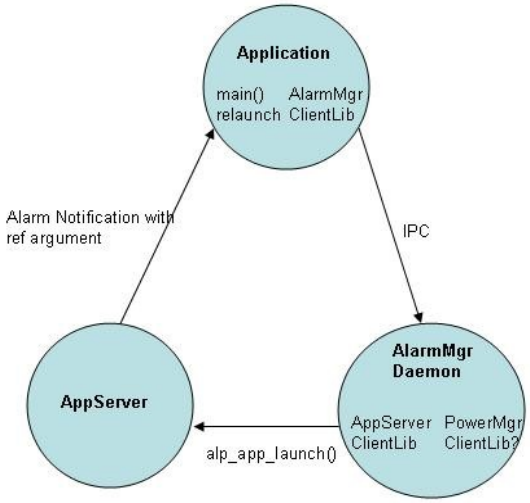
The Alarm Manager:

- Keeps track of when the device should next be woken up, if power management is likely to put the cpu to sleep.
- Calls alp_app_launch() to launch the applications for which an alarm event is due.
- Supports more than one alarm per application.
- Stores the alarm database using SQLite for persistence.

The Alarm Manager doesn't have any UI of its own; applications that need to bring an alarm to the user's attention must do this through the Attention Manager. The Alarm Manager doesn't provide reminder dialog boxes, and it doesn't emit the alarm sound. Think of it as “alarm clock” not “burglar alarm” - perhaps the name should have been “time reminder manager”.

Architecture

The following is the overview of the Alarm Manager architecture:



Attention manager

> The Attention Manager manages system events that need to be presented to the user, typically in the form of a dialog box asking for acknowledgment. When some interesting event happens (like an incoming phone call), Attention manager is used when you want to tell the user about it; Notification Manager is used when you want to tell a program about it. The Attention Manager uses a priority scheme to manage presentation of multiple simultaneous items. A phone call is more urgent than a calendar event.

The Attention Manager is a standard facility by which applications can tell the user that something of significance has just happened. The Attention Manager does not generate

these events. It is a collection point for them, and a handy standard way to tell the user about them. The “collection point” part is GUI-neutral; the “tell the user” part is necessarily tied to a GUI.

The Attention Manager provides a single alert dialog, and maintains a list of all "alert-like" events. Together these get the user's attention when needed, and allow the user to deal with the attention event or dismiss it for later review. This avoids forcing the user to click through a series of stale alert dialogs (as seen in many PC-based calendar programs). Often the user won't care about some missed appointments or phone calls - although he or she may care about others. With the Attention Manager, the user can selectively dismiss or follow up on the alert events, and not have to deal with each alert dialog in turn.

Applications have complete control over the types and level of attention they can ask for.

Typical flow of an attention event

- An application (e.g. the calendar app) requests the Alarm manager to wake at some time in the future.
- When that future time is reached, the Alarm Manager sends an event to the application. (Alarm manager could, but doesn't, use the notification manager for this. It knows which app asked for the alarm, and it can deliver the event directly by asking the app server to start the application and tell it *“you have been started to handle an alarm”*). When started or restarted, the application will post an event to the attention manager with the appropriate priority.
- The Attention Manager will present an alert dialog based on the event type and priority.
- The Attention Manager's sole duty is to interact with the user when an event must be brought to the user's attention.

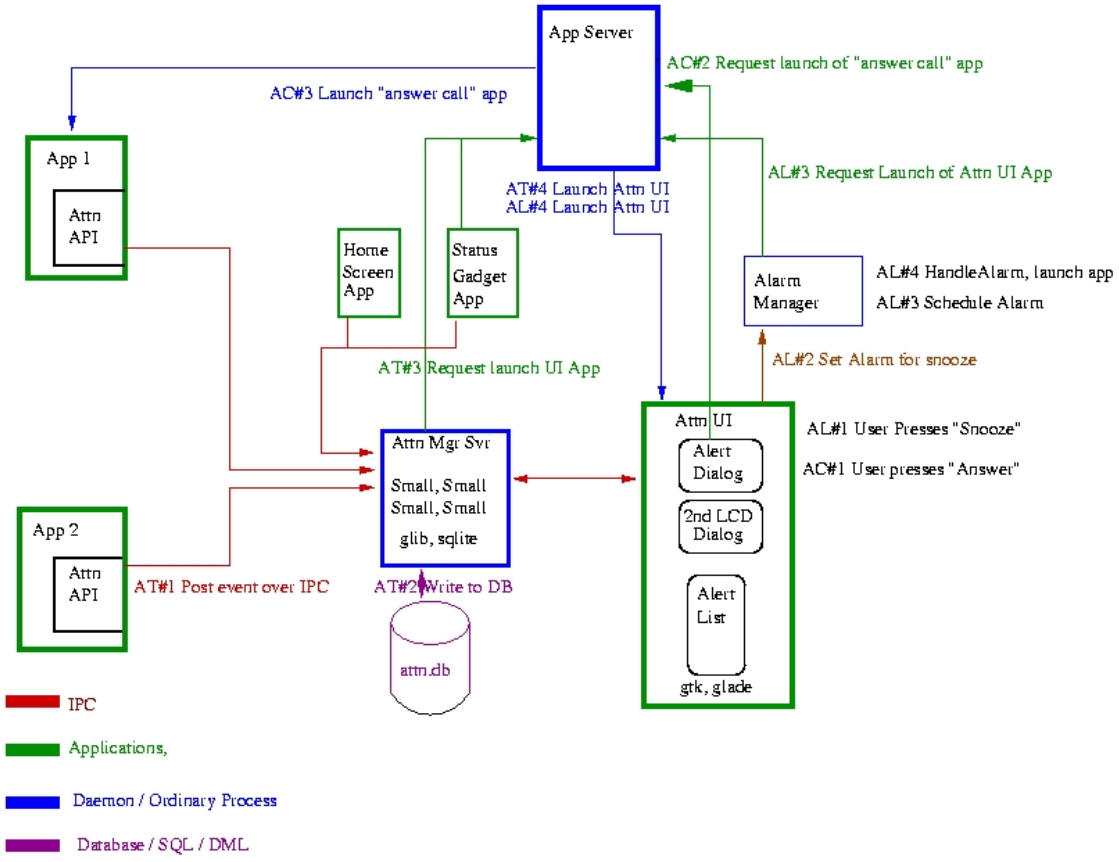
When the Attention Manager isn't appropriate

The Attention Manager is designed to remind the user about a restricted set of attention events. The Attention Manager doesn't replace error messages. Applications should use modal dialogs and other existing GUI and OS facilities to handle these cases.

The Attention Manager is also not intended to replace the ToDo (Tasks) application, or act as a universal inbox. Applications must make it clear that an item appearing in the attention manager is simply a reminder, and that *dismissing the dialog does not change, delete, or cancel the item itself*. That is, saying "OK" to an attention message does not delete the appointment, and dismissing an SMS reminder does not delete the SMS message from the SMS inbox. Dismissing the event does clear the event message from the attention manager.

The Attention Manager is not a general event logger or event history mechanism. It contains only active attention events.

Architecture



The Attention Manager meets these design goals:

- separate the UI from attention event tracking mechanisms
- make it easy to change the appearance and behavior of the posted events
- support persistent storage of events, that will survive a soft reset
- high performance (you can't be slow to inform the user of an incoming call)

The code is in directory servers/attnmgrd.

Exchange manager

The Exchange Manager is a central broker to manage data flow between applications or between devices. A request to the Exchange Manager has a verb (“get”, “store”, “play”), an object (parameters used to identify the specific item to be acted on), a datatype (in the form of mime-type), and possibly a destination (e.g. another device).

Typical uses of the Exchange Manager include beaming a contact to another device; taking a picture using the camera from an MMS application; looking up a vCard based on caller ID; viewing an email attachment, etc.

The Exchange Manager is extensible: new handlers can be created for new data types and actions as well as for new transports (e.g. IR, local, Bluetooth, SMS, TCP/IP, etc.)

Most applications need to be able to perform different tasks (such as play, get, store, print) on several types of data. Exchange Manager offers a simple API that lets any application delegate the handling of some data to whichever component declared the capability for this action/type-of-data pair. In addition, the destination of the request (where the action will actually be performed) can be specified as the local device or a remote location. This opens up new possibilities such as directly sending a vCard to someone through Internet while being on the phone with them.

Features

Exchange Manager enables an application or system component to request some action on some data, without knowing details of who will fulfill the request. In addition, the client can request that the action be performed either on the local device or on some specific other destination (phone or desktop PC), using any available transport.

An application with a service it wants to make available to others, registers a handler to tell the system it can do this specific action on this specific type of data. The handler may also specify that it will accept only Local requests, or that it will accept all requests. Each registered handler is valid for exactly one combination of action/data type. The action/data type is defined by a verb, and a MIME type (e.g. 'store' - 'text/vCard').

Transport modules are responsible for carrying a request from the source to the destination device. Transports are quite independent and can be added or removed at any time. It is unlikely though that many will be added by third parties as other libraries will provide Local, IR, BT, SMS, and TCP. Non-ALP destination systems can be compatible with Exchange Manager by running an exchange manager daemon and transports.

When a request arrives at the destination (which may be the local machine), the transport hands it to the exchange manager, who will then invoke the corresponding action handler to do the work. A result may be sent back to the initiator. It is therefore possible to use this Exchange Manager to retrieve some data (not just send it), or pass some data and get it back modified in some way. A use case would be using your phone to retrieve a contact in your desktop PC address book, or get a contact's photo and name given the phone number.

Handlers can be registered or unregistered at any time. A board game might register the 'moveplayer' action (a handler to receive other players moves) when it is launched, and

would send its own moves by requesting this same action from the other user device. The game would unregister the handler when it quits. Note: this example does not mean exchange manager could be used as a network hub to handle more than peer-to-peer exchanges.

An application cannot verify the availability and identity of a handler (one goal of the Exchange Manager to enable a registered-but-otherwise-unknown handler execute a request).

A non-exhaustive list of verbs includes 'store', 'get', 'print', 'play'. Verbs can be accompanied by parameters. Only a few parameters are common to all verbs (e.g. a human readable description of the data that may be used to ask the user if he accepts what he is receiving). Parameters are passed as a tag/value pair (the value being int or string). Most parameters are optional and depend on the specific handler definition. Parameters allow the result of an action handler to be precisely customized to the client needs. Parameters can be used by the action handler to find out how exactly it should perform its action, or by the transport module to get the destination address or other transport specific information.

An Exchange Manager daemon is started at boot time. The daemon listens for incoming action request for all transports. When a transport has an incoming request ready, the Exchange Manager dispatches it to the right action handler. The action handler then performs its duty, and returns the result back to the transport. The answer is then sent back to the originator.

When the originator is Local, the user is never asked to accept the incoming data. When the originator is remote, it is transport-dependent whether a user confirmation is asked or not. For IR, it is assumed that the user accepts implicitly when directing the device toward the emitter. For Bluetooth, it depends on whether the connection is paired or not. For SMS, the mobile network identifies the originator (in addition, there is no 'connection' with SMS). For TCP, the transport configuration will tell whether the originator IP is authorized, and it will ask if not.

For handlers that are essentially data consumers and don't return anything (like 'store'), multiple handlers may register for the same verb/data. For this reason, it is a handler can specify at registration time if it must be unique or not.

An AlpExgRequest is the Exchange Manager data structure an application works with. It is an opaque structure that contains all the information characterizing a request: the verb, the parameters, the data reference and the destination. There are APIs to set and get all of them. The data itself can be specified in multiple ways: file descriptor (data will be read from this fd by the transport) or URL (URL is sent, and action handler will access data through the URL).

- There are a number of in-the-box PIM applications that provide default handlers for published standard services (like Bluetooth connections). A third party application may try to register a duplicate handler for a service. If the default handler declared it wanted to be unique, the user is alerted and asked to choose which application should be the installed handler. This approach has been designed to be independent of whichever handler happens to be installed first. We intend that all preinstalled applications be signed and be trusted.
- When there is a non-authenticated incoming connection, the user is asked to authorize it. Local connections are always initiated by the user and are always valid. IR is considered authenticated, as the user must explicitly direct his device to the initiator. Paired Bluetooth is authenticated by definition. SMS is authenticated by the mobile network. TCP is considered authenticated if the source IP address is in the table of trusted sources (although this may be revisited as it is possible to spoof the source IP). TCP may be configured to require a challenge password before it accepts a connection.
- Above connection level authentication, handlers may also require permission from the user before they perform their action. This is handler specific. 'get vCard' is an example where user authorization may be requested.

ALP IPC

The ALP IPC service provides a lightweight, robust, message-based IPC mechanism. The implementation is based on POSIX sockets, but it can be layered on other IPC mechanisms (such as named pipes, DBUS, etc) instead. ALP IPC has a peer-to-peer architecture that minimizes context switches, an important feature on embedded architectures like ARM (this architecture caches virtual addresses, and so flushes the CPU cache on each context switch). ALP IPC provides a simple and preferred mechanism used in the implementation of the Hiker framework, but all other POSIX IPC mechanisms are still available.

The Hiker framework makes extensive use of IPC calls. Framework users (application programmers) will link to client side libraries which encapsulate the IPCs and make the framework services accessible as local procedure calls.

The API can be seen in the Doxygen at doc/html/alp__ipc_8c.html.

Each server process exchanges messages with one or more clients, using ALP IPC. The server process creates an AlpChannel. Clients connect to the channel and receive an AlpConnection pointer that they can use to send/receive messages to the server. The format of the messages is completely up to the server and clients of the channel. Sending an IPC is thus a four step process, using these functions:

1. `AlpChannel* alp_ipc_channel_create (const gchar *channelName, AlpIpcChannelAccessMode accessMode)`

2. `AlpConnection* alp_ipc_channel_connect (const gchar *channelName, AlpConnectionUsageHint usageHint, AlpChannelConnectCB callback, gpointer cbData)`
3. `AlpMessage* alp_ipc_message_create()`
4. `alp_status_t alp_ipc_connection_send (AlpConnection *connection, AlpMessage *message)`

There are several other APIs, e.g. to retrieve the pid, group id and user id of the other end of the connection. That information is used by the security framework to determine if the requester is authorized to make that request.

Communication can be synchronous or asynchronous. There are also APIs which allow client applications to register for asynchronous callbacks from a server. When done asynchronously, processes receive messages via a callback mechanism that works through the gLib main loop, and makes the callback when the GUI program is otherwise idle. So this part of ALP IPC is tied to the GUI framework (GTK+ in this case).

9. Security Policy Framework

The Security Policy Framework (SPF) is the component which controls the security policy for the device. The security policy specifies which kinds of applications (from PalmSource, from the handset vendor, from the carrier, from third parties) can access which kinds of files and resources. The policy can be created by a carrier or manufacture and can be updated. Policy is flexible and separate from the mechanisms used to enforce it.

Typical elements of a policy address use of file system resources, network resources, password restriction policies, access to network services, etc. Each policy is a combination of these attributes and is tied to a particular digital signature. Applications are checked for a digital signature (including no signature or a self-signature) and an appropriate security policy is applied to the application. One of the policy decisions that can be made by the framework is whether the user should be consulted – this allows for end-users to control access to various types of data on the device and ensure that malicious applications will not access this data covertly. Other types of decisions are allow/deny which may be more appropriate for a carrier to use to protect access to network resources, etc.

The ALP LSM (Linux Security Module) is a kernel level enforcement component that works in concert with the SPF. The LSM controls the actual access to files, devices and network resources. We intend to open source our LSM, but it is not user-level code, and is not included with the Hiker framework.

The Hiker framework offers some user control over who is allowed to connect to the user's device and request action from it.

10. Trademarks etc

Access is a registered trademark of Access Co Ltd.

UNIX is a registered trademark of The Open Group

Palm OS is a registered trademark of Palm Trademark Holding Company, LLC.

Mac is a registered trademark of Apple Computer Inc.

Linux is a registered trademark of Linus Torvalds.
