The Hiker Project: An Application Framework for Mobile Linux Devices

David "Lefty" Schlesinger
ACCESS Systems Americas, Inc.
lefty@{hikerproject.org,access-company.com}

Abstract

The characteristics of mobile devices are typically an order of magnitude different than desktop systems: CPUs run at megahertz, not gigahertz; memory comes in megabytes, not gigabytes; screen sizes are small and input methods are constrained; however, there are billions of mobile devices sold each year, as opposed to millions of desktop systems. Creating a third-party developer ecosystem for such devices requires that fragmentation be reduced, which in turn demands high-quality solutions to the common problems faced by applications on such devices. The Hiker Project's application framework components present such solutions in a number of key areas: application lifecycle management, task-to-task and task-to-user notifications for a variety of events, handling of structured data (such as appointments or contact information) and transfer of such data between devices, management of global preferences and settings, and general security issues. Together, these components comprise an "application framework," allowing the development of applications which can seamlessly and transparently interoperate and share information.

ACCESS Co., Ltd., originally developed the Hiker Project components for use in their "ACCESS Linux Platform" product, but recently released them under an open source license for the benefit and use of the open source community. This paper will describe, in detail, the components which make up the Hiker Project, discuss their use in a variety of real-world contexts, examine the proliferation of open source-based mobile devices and the tremendous opportunity for applications developers which this growth represents.

1 The Need for, and Benefits of, a Mobile Framework

The typical cell phone of today is generally equivalent in power, in various dimensions, to a desktop system of four or five years ago. Cell phones increasingly have CPUs running at close to half a gigahertz, dynamic RAM of 64 megabytes and beyond, and storage, typically semiconductor-based, in capacities of several gigabytes. Current MP3 players can have disk capacities well beyond what was typical on laptop systems only two or three years ago. The usage models for cell phones and other mobile devices, however, tend to be very different than that of desktop systems.

The bulk of applications-related development effort on open source-based systems has been primarily focused on servers and desktop devices which have a distinct usage model which does not adapt well to smaller, more constrained mobile devices. In order to effectively use open source-based systems to create a mobile device, and particularly to enable general applications development for such a device, a number of additional services are needed.

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. Linux's `fork()` and `exec()` calls; C programs have an entry point called "main" that takes a couple of parameters and returns an integer; etc.).

When we use the term "application framework," we intend it to 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, full keyboards, and general pointing devices. A cell phone typically has a screen fewer than three inches diagonal (although pixel resolutions

can be equivalent to QVGA and higher), a complement of under twenty keys and a "five-way" pointing device or a "jog wheel." Based on experience refining Garnet (formerly known as "Palm OS"), we believe that there should be only one active window on a typical handheld device. As another example, when the user starts a new application, the previous application should automatically save its work and state, and then exit.

Similarly, the optimal conceptual organization of data is different on mobile devices. Rather than an "application/document" paradigm, using an explicit, tree-structured file system, a "task-oriented" approach, where data is inherent to the task, is more natural on these devices. Tasks, as opposed to applications, are short-lived and focused: making a call, reading an SMS message, creating a contact or appointment, etc. Occasionally, tasks will be longer-lived: browsing the web, or viewing media content. Another typical attribute of such devices is regular, unscheduled interruptions: a low-battery warning, an incoming email, a stock or news alert.

As well as the task of managing the lifecycle of programs (launching, running, stopping), the application framework must also 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 known as a "bundle." Bundles are convenient for users and third party developers, and allow software to be passed around and downloaded as an atomic object.

A third task of the application framework is to support common utility operations for applications, such as communication between applications, keeping track of which applications handle which kind of content, and dealing with unscheduled events 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. For example, 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.

The Hiker components then, broadly speaking, focus on several key areas:

- presenting a common view of applications to the user (*Application Manager* and *Bundle Manager*),
- communicating time-based and asynchronous events between applications or between an application and the user (*Notification Manager*, *Alarm Manager*, *Attention Manager*);
- interoperability of applications and sharing of information between them (*Exchange Manager*, *Global Settings*), and
- performing these functions in a secure context (*Security Policy Framework/Hiker Security Module*)

The *Abstract IPC* service is used by these components in order to simplify their implementation and allow them to generally take advantage of improved underlying mechanisms through a single gating set of APIs. Taken together, they provide mechanisms to allow seamless interoperability and sharing of information between suites of applications in a trusted environment.

These components are intended to offer several concrete benefits to the development community:

- They provide real reference implementations that can serve as the basis for application developers who want to write interoperable applications for mobile devices.
- They (hopefully) help to jump-start activities related to mobile devices in several key areas (e.g. security) by filling a number of current gaps in the services available to applications in that space.
- They can help to generally increase interest in application development for open source-based mobile devices
- They might encourage other companies to both participate in these projects and to contribute new projects as well.

1.1 Access to the Project, Licensing Terms, etc.

Full source code and other reference material for Hiker can be found at the Hiker Project web site, `http://www.hikerproject.org`.

Sources are currently available as tarballs, but we expect to be putting a source code repository up in the near future. Several mailing lists are available, and a TRAC-based bug reporting/wiki system will be in place shortly. A large amount of detailed API documentation, generated by Doxygen, is also available on the site.

Hiker is dual-licensed under the Mozilla Public License, v. 1.1 and the Library General Public License, v. 2, with the exception of the LSM-based portion of the Security Framework, which, as an in-kernel component, is licensed under GPL v. 2.

2 The Application Manager

The Application Manager controls application lifetime and execution and is the component that is responsible for maintaining the simple application task model that users expect on a phone. The application manager starts and stops applications as appropriate, ensures that the current running application controls the main display, maintains a list of running applications, places applications in the background, and prevents multiple launches of a single application.

The Application Manager is initiated at system start-up and provides the following services:

- Application launching mechanism and management of application lifecycle.
- Routing of launch requests to the existing instance (if the application is already running).
- Coordination of the "main UI app" – retires the current application when a new one is launched, and launches a default when needed.
- May also provide default behavior for certain important system events (i.e., hard key handling, clamshell open/close).

The Application Manager runs in its own process. Applications typically run in their own processes and control their own UI. This simple mapping of applications to processes provides a secure, stable model for application execution. To maximize utility on small screen form factors, the Application Manager will preserve the standard "Garnet OS" behavior of having one main UI application at a time, with that application having drawing control of the main panel of the screen (exclusive of status bars, etc.) When the user runs a new application, the system will generally ask the current one to exit (although there is a facility for applications which need to continue execution in the background). Also, as in legacy "Garnet OS" and desktop operating systems like Mac OS X, only a single instantiation of any given application at time will be running.

The Application Manager handles the high-level operations of launching applications, and provides a number of APIs to applications and to the system for access to these services. It maintains a list of currently running applications, and keeps track of which one is the "primary" or "main UI" application. This special status is used to coordinate the display and hiding of UI when the user moves between applications. Note that an application that is not the main UI application may still put up UI under exceptional circumstances, it is simply recommended that this be done only occasionally (for example to ask the user for a password), and that it be in the form of a modal dialog. The user's attention is generally focused on the main UI application, and so UI from background applications is often an interruption. It is expected that most applications will not run in background mode.

3 The Bundle Manager

The Bundle Manager combines a file format for distributing single files that contain applications and all their dependencies (application name, icon, localized resources, dependant libraries, etc.) along with functions for installing, removing, and enumerating application "bundles." The Bundle Manager takes care of allocation of per-application storage locations in writable storage in the file system, and providing access to localized resources contained within the bundle. Through the Bundle Manager enumeration mechanism, multiple application types are merged and can be launched through a common mechanism.

The Bundle Manager is the system component responsible for controlling how applications, and supplemental data for applications (libraries, resources, etc.), are

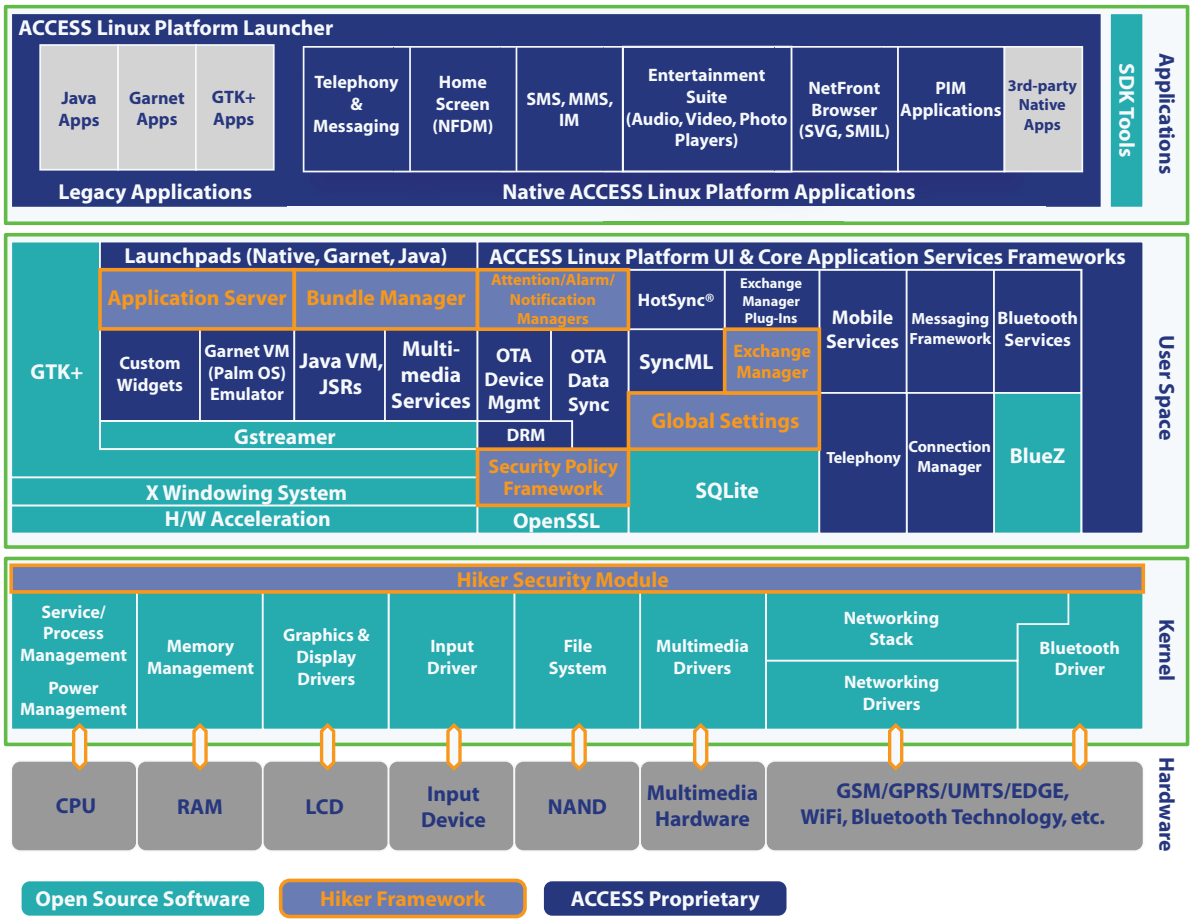

Figure 1: *The Hiker Components in the ACCESS Linux Platform*

loaded onto a system using the Application Manager and other framework components, manipulated, transmitted off of the system, and removed.

The Bundle Manager is a mid-level library and server which provides easy access to application resources for developers, as well as maintaining state about bundles present in the system.

The Bundle Manager is designed around the notion of "bundles" as concrete immutable lumps of information which are managed on a device, where each bundle can contain an arbitrary amount of data in a format appropriate to that bundle. Each bundle type is defined both in terms of how it is stored on the device, and as a "flattened" format suitable for transmission as a stream out of the device. (These formats may be the same, different, or overlap at various times).

The Bundle Manager is the channel through which all third-party applications are distributed and loaded on to a device. The server component of the Bundle Manager is intended to be the only software on the device which has permission to access the bundle folder on the internal filesystem, requiring interaction with the Bundle Manager for installation or removal.

The Bundle Manager provides consistent mechanisms for retrieving resources, both localized and unlocalized (i.e., loading the files or other bundle contents, perhaps in a localized folder-name) from bundles.

4 The Notification Manager

Notification Manager provides a mechanism for sending programmatic 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 ap-

plications that are registered to receive them but are not currently running.

Notifications are general system level or user application level events like application installed/uninstalled, card inserted/removed, file system mounted/unmount, incoming call, clamshell opened/closed, time changed, locale changed, low power, or device going to sleep/waking up. The Notification Manager has a client/server architecture.

4.1 Notification Manager server

The Notification Manager server is a persistent thread in a separate system process which keeps track of all registered notifications and broadcasts notifications to registered clients. The Notification Manager server also communicates with the Package Manager and Application Server.

4.2 Notification Manager client library

Client processes call APIs in the Notification Manager client library to

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

The Notification Manager client library uses the Abstract IPC framework to communicate with the Notification Manager server.

4.3 What the Notification Manager is not

1. The Notification Manager should not be used for application specific or directed notifications like alarms or find.
2. 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).

5 The 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 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:

- Works with power management facilities to register the first timer;
- Calls the Application Manager to launch the applications for which an alarm is due;
- Supports multiple alarms per application; and
- Stores its alarm database in SQLite for persistence.

The Alarm Manager has no 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 play the alarm sound. Applications should use the Attention Manager to interact with the user.

6 The Attention Manager

The Attention Manager manages system events that need to be presented to the user, such as a low battery condition or an incoming call (rather than programmatic events delivered to other applications like the service provided by the Notification Manager). The Attention Manager uses a priority scheme to manage presentation of items needing attention to the user. The infrastructure used by applications and system services to ask for attentions and the storage of the currently pending list of events requiring the user's attention is separate from the actual presentation of the events to the user.

The Attention Manager is a standard facility by which applications can tell the user that something of significance has occurred. The Attention Manager is responsible only for being a nexus for such events and interacting with the user in regards to these events; it is not responsible for generating the events.

The Attention Manager provides both a single alert dialog and maintains a list of all "alert-like" events. Together these improve the user experience by first getting the user's attention when needed, then allowing the user to deal with the attention or dismiss for review later. By handling it this way, it is no longer necessary to click through a series of old alert dialogs. Often the user doesn't care about most of the missed appointments or phone calls—although he might care about a few of them. Without the Attention Manager, the user cannot selectively dismiss or follow up on the alert events but would instead 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 of "Date Book") requests the Alarm Manager to awaken it at some time in the future.
- The Alarm Manager simply sends an event to an application when that future point in time is reached. The application can then post an event to the Attention Manager with the appropriate priority.
- The Attention Manager will present the appropriate alert dialog based on the event type and priority.
- The Attention Manager is designed solely to interact with the user when an event must be brought to the user's attention.

6.1 When the Attention Manager isn't appropriate

The Attention Manager is specifically designed for attempts to get attentions that can be effectively handled or suspended. The Attention Manager also doesn't attempt to replace error messages. Applications should use modal dialogs and other existing OS facilities to handle these cases.

The Attention Manager is also not intended to act as a "To Do" or "Tasks" application, nor act as a "universal in-box." Applications must make it clear that an item appearing in the Attention Manager is simply a reminder, and that dismissing it neither deletes nor cancels the item itself. That is, saying "OK" to an attention message regarding an upcoming appointment does not delete the appointment, and dismissing an SMS reminder does not delete the SMS message from the SMS inbox.

The Attention Manager is not an event logger, nor an event history mechanism. It contains only the state of active attention events.

The Attention Manager is organized in order to meet specific design goals, which are:

- To separate UI from attention event logging mechanisms;
- To provide sufficient configurability so that a licensee may alter both the appearance and behavior of the posted events;
- To maintain persistent store of events that will survive a soft reset;
- To be responsive (need to be quick from alert to UI—prime example is incoming call); and
- To minimize memory usage (i.e. try to get as small of memory foot print as possible).

The server model currently utilized relies on an init script to start a small daemon. The daemon accepts IPC requests to post, update, query or delete events and maintain such event state in a database to provide persistent storage for the events. The daemon launches the attention UI application which will display the appropriate alert dialog. If the snooze action is chosen for an event (provided that a snooze action is associated with the event), the Attention UI application will call the Alarm manager to schedule a wakeup. The wakeup will be in the form of a launch or relaunch via the Application server. In this model, the attention status gadget is assumed to be polling the Attention Manager daemon for active event state and using that information in displaying status. If the attention status gadget is "clicked" on, it starts the Attention UI through the Application Server.

6.2 Features

The Attention Manager implements the following major components:

- attention events;
- an API library for posting, updating, deleting, and querying events;
- a server through which events are posted, retrieved and managed;
- a UI application that displays the alert dialogs;
- an event database and associated DML API; and
- a status gadget for the status bar.

7 Abstract IPC

The Abstract IPC service provides a lightweight, robust interface to a simple message-based IPC mechanism. The actual implementation can be layered on top of other IPC mechanisms. The current implementation is based on Unix sockets, but this mechanism can be layered on other IPC mechanisms if required. The current implementation has a peer-to-peer architecture that minimizes context switches, an important feature on some popular embedded architectures.

The Abstract IPC Service comprises an API for a simple interprocess communication (IPC) mechanism used by the framework components described in this paper.

The goals of this design include:

- Independence from underlying implementation mechanisms (e.g. pipes, sockets, D-BUS, etc.);
- A simple, easy to use send/receive message API;
- Support for marshalling/unmarshalling message data; and
- Minimization of context switches by sending messages directly between processes without passing through an intermediary process.

The IPC mechanism is based on a single server process exchanging messages with one or more clients. The server process creates a channel; clients connect to the channel and receive a connection pointer 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.

Communication can be synchronous or asynchronous. When done asynchronously, processes receive messages via a callback mechanism that works through the gLib main loop.

8 Security Policy Framework and Hiker Security Module

The Security Policy Framework (SPF) is the component which controls the security policy for the device. The actual policy used by the framework is created by a licensee and can be updated. Policy is flexible and separate from the mechanisms used to enforce it. The policy can express a wide (and extensible) range of policy attributes. 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 Hiker Security Module is a kernel level enforcement component that works in concert with the Security Policy Framework. The Security Module controls the actual access to files, devices and network resources. Because it is an in-kernel component, the Security Module is released under the GPL.

There must be some user control on who is allowed to connect to the user's device and request action from it. Security is mostly based on the user control at the following levels:

- There will be default handlers (implemented, for example, by in the box PIM applications) for a bunch of published standard services. In the event where a third party application would try to register a duplicate handler, and the first handler declared it wanted to be unique, the user would be alerted

and asked to arbitrate which application should be the installed handler. The user is the authority that tells the system which handler wins in case of conflict. This security works whichever handler installs first. In a model where all installed applications are signed and therefore trusted, we can rely on what the application do when they register.

- When there is a non-authenticated incoming connection, the user is asked to authorize the connection. Local is obviously initiated by the user and is 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 may be considered authenticate if the source IP address figures in the table of trusted sources (although this may be discussed as it is easy to spoof the source IP). TCP may also be configured to require a challenge password before it accepts to read from the connection.

- Above the connection level authentication, handlers may also require permission from the user before they perform their action. This is handler specific. "get vCard" is obviously a very good candidate to user authorization.

9 The Exchange Manager

The Exchange Manager is a central broker to manage inter-application/inter-device communication. Requests to the Exchange Manager contain verbs ("get", "store", "play"), data (qualified by mime-type) as well as other parameters used to identify the specific item to be affected by the request. Use cases 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 an extensible framework: 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.).

There is a need for any application to be able to perform different tasks (such as play, get, store, print, etc.) on several types of data. This component offers a simple, yet expandable, API that lets any application defer the actual handling of the data to whoever declared he was the handler 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 universal possibilities such as directly send a vCard to someone through Internet while being on the phone with him.

This component also implements the legacy Palm OS "Garnet" Exchange Manager functionality (send data to a local or remote application). This simply corresponds to the action *store*

The Java VM also implements a similar functionality (JSR211—CHAPI). It is foreseen that the Java and native components will interoperate. In other words, a Java application will be able to send a vCard to a remote device through IR, as well as receive data from a local native application, for some examples.

9.1 Features

The purpose of this component is to enable an application or system component to request the system to perform some action on some data, without knowing who will carry and 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 (mobile phone or desktop PC, etc.), using any available transport. This opens up new interesting scenarios that were not possible to do before.

An application that implements a service it wants to make available to others registers a handler to tell the system it can perform this specific action on this specific type of data. The handler may also specify that it will accept only local request, or that it will accept all requests. Each registered handler is valid for only 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 to carry the request from the source to the destination device. When the request arrives at the destination (which may be the local machine), the transport will hand it to the Exchange Manager who will then invoke the corresponding action handler to do the actual work. A result may be sent back to the initiator, which means it is also possible to use this component to retrieve some data (not only send

it), or pass some data and get it back modified in some way. An obvious use case would be to use your phone to lookup a contact in your desktop PC address book, or retrieve a contact's photo and name given its phone number.

Handlers can be registered or unregistered at any time. A board game would 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 that this example does not mean Exchange Manager could be used as a network media to handle more than peer to peer exchanges.

An application cannot verify the availability and identity of a handler and this would not make sense in the general usage (it is the goal of the Exchange Manager to be able to have an unknown handler execute a request). If an application needs to authenticate the handler, then this means it looks for a very precise handler it knows and in this case, it could use, for example, data encryption to ensure only the person with the right key can understand the request.

Transports are independent modules that can also be added or removed at any time. Typical transports would include Local, IrDA, Bluetooth, SMS, and TCP. Non-Hiker destination systems must run at least an Exchange Manager daemon and transports as well. Except for the handler invocation part, this should be the same code for any Linux platform (the sharedlib, daemon and transports only use standard Linux and GTK services). It would be easy, for example, to implement a new encrypted transport, should the need arise. The library provides a standard UI dialog to let the user select a transport and enter the relevant parameters. If the transport information is missing in the request, the Exchange Manager will itself pop this UI up at the time it needs the information. If the transport determines that the destination address is missing, it will also popup an address selection UI.

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). Some are mandatory but most are optional and depend on the specific handler definition. Using parameters, the result of an action handler can be precisely customized to the client needs.

A non-exhaustive list of actions that will be defined include *store*, *get*, *print*, and *play*.

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 (or at any other time). The daemon is used to listen for incoming action request for all transports. When a transport has an incoming request ready, the daemon 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, whether a user confirmation is required will depend on the transport being used. For IrDA, it is assumed that the user implicitly accepts as he directs his device toward the emitter (it is still to be decided if we ignore the possibility that someone would be able to beam something to you without your consent while you have the device turned on). For Bluetooth, it depends 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"), it may make sense to let multiple handlers register for the same verb/data. For this reason, it is left to the handler to specify if it must be unique or not at registration time. In case it must be unique and someone tries to register a second handler for the same verb/data, the user would be notified and he would have the responsibility to arbitrate which of the two handlers should be the active one.

To maintain transfer compatibility with phones or legacy Garnet OS devices, the initiator may set a parameter to tell he wants to use Obex as the transport. In this case, the only verb allowed is "store." The transport plug-in will recognize this parameter and send the data using the OpenObex daemon. As well, a transport plug-in can also receive data from the OpenObex daemon. It would

treat that like an incoming connectionless "store" action on the received data.

For all other combinations of action/data type, the transport protocol is ours (or third party in case of third party transport). Obex protocol is handled by the OpenObex library. SMS NBS transport is handled in the SMS component. Some specific Bluetooth profiles (e.g. basic imaging profile) could also be handled by the Bluetooth transport in order to maximize compatibility with other kinds of devices.

An "Exchange request" is the only entity 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).

10 Global Settings

The Global Settings component provides a common API and storage for all applications and services to access user preferences (fonts, sizes, themes) and other application settings and configuration data. Global Settings are hierarchical and could be used, e.g., as the basis for OMA Device Management settings storage. The component is designed to support the security requirements of OMA Device Management. The storage for Global Settings uses the recently open-sourced libsqlfs project layered on SQLite. Global Settings provides generic, non-mobile specific, storage.

The Global Settings service provides functions for storing user preferences. It provides APIs for the setting and getting of software configurations (typically key-value pairs such as "font size: 12 points"). 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. `applications/datemanager/fontsize`. Indeed, Global Settings is actually implemented on top of an abstract POSIX file system: each key has a value and meta data such as a user id, a group id, and an access permission.

To accomplish this, Global Settings utilizes libsqlfs, a service component which creates the abstraction of a POSIX file system within an SQLite database file. An initial implementation was attempted using gconf. To support OMA Device Management requirements, we dropped the gconf design, which could not provide security over keys.

The Global Settings service has two parts: the daemon (aka server) and the client library. You could store settings by simply having a server process. However, to make it trivial for the clients to talk to the server, we also provide a client library which handles the IPC to the server. The client library is linked in with the client application.

The client library and the daemon communicate via Abstract IPC. The daemon does the actual I/O for the data; it links with the SQLite library and does the key content reads and writes on SQLite databases, with the tree hierarchy actually 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 everyone else. 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 will be only readable and writable by the daemon process.

10.1 Data Model

1. We expose key/value pairs where values amount to the "contents" of the key and can be arbitrary data of arbitrary size.

2. Key descriptions (in multiple languages) are supported only by convention (see below).

3. We support settings that are user, group, or 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.

The possible key names or key strings form a key space, which is similar to the space of file paths. The key strings are also called key paths. Each key path can be absolute or relative; absolute key or key paths must start with /, and relative key paths are relative to a "current directory" or a "cwd." There are APIs to get and to set the "cwd."

A "directory" is a key which contains other keys. A directory is similar to the interior nodes in the OMA Device Management Tree definition. To simplify our design, we disallow a directory key from having its own content. So if you add a child key to an existing key which has no value (or having the null value), that key becomes a directory and attempts to set the content for that key will fail in the future. Permissions for directories thus follow the semantics of POSIX file system.

A directory key can be created in two ways:

- A directory is created if a request is made to create a key that would be a child or a further descent of the directory
- Or a key can be created with an explicit API for this purpose

Key names 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 Device Management.

Some typical keys are represented below:

- `/dm` for OMA Device Management
- `/capabilities` for "is java installed," "is Garnet OS emulation installed," etc.
- `/packages/com.access.apps.app1` for preferences of "app1"

The Global Settings service is not meant for storage of security-sensitive data such as passwords or private keys. The data in the Global Settings are only protected by POSIX file permissions and are not encrypted or otherwise protected!

Global Settings implements the POSIX file permission (user, group and others, readable, writeable or executable) model on the keys and the key hierarchy. So a run time process has to acquire the appropriate user or group IDs to access a key the same way it access a file with the same POSIX permission bits in the host file system.

Otherwise, Global Settings places no meanings on the uids and gids, except the uid 0 which has permissions for everything, as root. The SUID bit is not honored and ignored for Global Settings.

The x bits in directories determine if the directories can be entered; the x bits in non-directory keys are ignored and have no meaning currently.

The uid and gids of keys are modifiable according to the following rules:

- The owner id cannot be changed except by the root.
- The group id cannot be changed except by the root or the owner.

Global Settings has reserved rooms for extra attributes for keys, currently not implemented but these could be used for access control lists or some other meta data but I treat such usage as the best to be avoided; if POSIX permissions provide all that's needed it is the best as it is simple and efficient.

With the above convention for keys, it becomes desirable for the applications or specific device (drivers) to create its key hierarchy during the installation time (or at the point of system image creation). Once created a hierarchy may be protected from access by applications other than the designated applications.

The initial key installation follow these steps:

- A group id is pre-assigned to a particular application (group). This assignment is to be guaranteed at run time by the Bundle Manager and system security.
- After an application is installed, it, or the Bundle Manager, will use a Global Settings tool or equivalent APIs to copy the key data from the default settings XML file into the Global Settings database and properly set the user and the group ids and the permissions for these keys.
- Later the Global Settings service ensures that the key hierarchy is accessible only to applications with the right group membership or the user id. The Unix file permission style permissions will be the only security mechanism protecting the key and key values; Global Settings does not provide encryption-based protection mechanism.

The Global Settings daemon is started at system boot time. Packages need to include their associated default preferences and these need to be installed in the correct fashion for the underlying preferences system.

Clients that wish to be informed of changes in settings can register a callback with the Notification Manager. The Global Settings daemon will send Notification Manager a string representing each key that it changes. Notification Manager will notify each application that has registered to be told about changes to that key. (Notification Manager will notify each application that registered for that key, or a prefix of that key. For example, an application 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 key change notification has the "notification type" (or "notify type") of a string of the form `/alp/globalsettings/keychange/` + the key string. For example, a change of the key `oma/apps/calendar/fontsize` will invoke a broadcast notification call with the notification type `/alp/globalsettings/keychange/oma/apps/calendar/fontsize`.

The Notification Manager currently 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 representative 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 key changes in the `/oma/apps/calendar/` key directory. The Notification Manager is the actual component which checks this and invoke the notification callback in client applications.

11 Conclusion

Mobile devices, and the applications running on them, represent a new frontier for open source developers. Worldwide cell phone sales are approaching one billion a year, with "smart phones" (that is, phones whose capabilities can be enhanced "post-platform," subsequent to their purchase by the consumer) showing the largest rate of growth. Open source-based operating systems have been of increasing interest as a foundation for work in the space.

By 2010, it's estimated that the number of "smart phones" sold per year will exceed the number of desktop systems. As many as half of those phones will be running Linux-based system, according to some analysts.

Linux and other open source software has taken on increased significance in this context. While a number of Linux-based phones have been shipped (primarily in China and other Pacific Rim geographies), few of them have been truly "open" systems, that is, the ability to develop software for and incorporate it onto such devices has been terribly limited.

Further, because the usage model for these devices is so different from the usual desktop-oriented paradigms, there are significant gaps in service-level functionality, particularly in areas such as abstraction of varying execution environments for the user, packaging of applications and related resources and metadata, information interchange between applications as well as between devices, and general approaches to security.

The security component, in particular, is key to the effort to foster a "third-party developer ecosystem" for mobile devices. Because of their very nature, as well as the regulatory structure within which they exist, certain operational characteristics of devices such as cell phones must be guaranteed. This is true even in the case of errant or malicious additional software being installed on the device.

A paper of this length can only provide a high-level overview of the components involved and their potential characteristics, usage, and interrelationships. A great deal more detailed information is available on the project web site.

Hiker is an attempt to address the most significant of the gaps discussed at the outset of this paper as a way of reducing fragmentation and encouraging participation in the mobile applications development context. The Hiker Project is intended to be a open, community effort. While ACCESS engineers are responsible for the initial implementation, the project is intended for the use of all developers interested in mobile applications development, and their participation in improving the framework is invited.

Proceedings of the Linux Symposium

Volume Two

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*