Promoting Innovative Mobile Applications via a Third Party IMS-Enabled Ecosystem

Ryan Skraba, Olivier Le Berre, Patrick Legrand, and Johann Daigremont

Today’s IP Multimedia Subsystem (IMS) client on a mobile device provides basic services, including voice, presence, messaging, and contact management. In the future, the IMS client should be prepared to handle a large variety of innovative and unexpected services that integrate into the client seamlessly among existing services. Bell Labs’ Client/Service IMS Architecture group has defined a product concept called API for Mobile IMS Client Application (AMICAL) that provides multiple levels of application programming interfaces (APIs) for third parties developing IMS-enabled applications. By using a plug-in framework, AMICAL allows users to enhance the IMS client with innovative features without modifying the proprietary core software. At another level of abstraction via an IMS kernel service, third party developers can write their own application completely independently of the IMS client, yet enable it with IMS features such as presence-enabled objects and easily deployed communication services. These components provide a demonstrably effective means to promote new and innovative IMS-enabled services and applications by third party developers. © 2008 Alcatel-Lucent.

Introduction

Bell Labs’ Client/Service IP Multimedia Subsystem (IMS) Architecture group is responsible for the development of innovative, engaging applications and services in an IMS network. One of the team’s first product concepts, released in 2005, was a full-featured IMS communication client now known as the Alcatel-Lucent 5440 IMS Windows Mobile* client. It provides, among other features, Session Initiation Protocol (SIP)-based Voice over Internet Protocol (VoIP), push-to-talk, instant messaging, presence services, and a synchronized network address book with advanced contact management.

This client serves as a basis for many other product concepts, including adding more communication features such as email, geolocation, user reachability, and network heterogeneity (the movement of the terminal through different radio technologies) [3]. To prevent the future IMS client from becoming fatter and less maintainable, a modular solution and application programming interface (API) was introduced to permit a graceful evolution of the terminal software. In order to enable new, innovative features as well as to benefit from third party applications development, a usable, flexible, and open client architecture is required.

The API for Mobile IMS Client Applications (AMICAL) architecture, shown in Figure 1, was developed for the IMS client to satisfy these needs. It includes two major components:
1. The plug-in framework, which permits tight integration of new features and enhancements into the user interface of the IMS communication client, and
2. The IMS kernel, which provides IMS functionality (such as communication and presence services) as an independent kernel service to applications independently of the IMS client.

These APIs and components not only simplify the internal development of the IMS client, but more importantly permit third parties to integrate into the IMS client or to add IMS communications to their applications.

In addition to enabling third party applications to become IMS-enabled, a future product concept called the dynamic service environment (DSE) is being researched to perform advanced application and service management for deploying these applications to the terminal, and controlling their life cycle to maximize their utility to the user. These product concepts are complementary and build on each other to provide an all-in-one terminal experience that benefits both the user and third party application developers.

**Plug-In Framework**

The AMICAL plug-in framework is delivered as part of the IMS communication client and is appropriate for developing third party features that do not require independent subsystems and processes or that do require tighter integration into the IMS client. These

---

**Panel 1. Abbreviations, Acronyms, and Terms**

AMICAL—API for Mobile IMS Client Application
API—Application programming interface
CID—Correlated ID
DAB—Dynamic address book
DAL—Dynamic application list
DLL—Dynamically linked library
DMD—Dynamic mobile desktop
DSE—Dynamic service environment
GSM—Global System for Mobile Communications
GUI—Graphical user interface
ID—Identifier
IMS—IP Multimedia Subsystem
IP—Internet Protocol
JSR—Java Specification Request
M2PUI—Machine-to-person user integration
PIIM—Predictive interactive IMS messaging
SIP—Session Initiation Protocol
URI—Uniform resource identifier
WLAN—Wireless local area network

---

**Figure 1.**

*Heavy client versus modular architecture.*

---

58 Bell Labs Technical Journal DOI: 10.1002/bltj
features are typically represented by a tab in the IMS client main interface and they share its process space and memory.

The plug-in framework permits innovative features to be dynamically deployed to the IMS client without modifying or recompiling the core software. As an example, the IMS client could be deployed to any device, but the push-to-talk video plug-in could be provided only to those with cameras.

**Architecture**

The plug-in framework is implemented in the high-level C# language for the Windows Mobile Pocket PC operating system. There is an open set of interfaces and base classes that cleanly separate the IMS communication client core and the third party plug-ins. Since they both depend only on the common interfaces, each can be developed, compiled, and managed independently, as shown in Figure 2.

The Mic2Access interface in the plug-in framework defines all of the functions and events that a third party plug-in can access in the IMS communication client. For the sake of clarity, it is divided into several smaller interfaces: AddressBookFunctions, GuiFunctions, SipFunctions, CommunicationFunctions, and (for research purposes) ExperimentFunctions. The actual implementation of these interfaces remains proprietary to the IMS communication client. As long as the interface remains backward compatible, the plug-in framework can evolve to include new functionality without breaking existing plug-ins.

A third party developer integrates a feature into the client by implementing a subclass of Mic2Plugin and giving it a unique identifier. An instance of this class is created dynamically when the application starts, and it is passed an implementation of the Mic2Access interface during initialization. At this point, the IMS communication client and all of the plug-ins can interact via the plug-in framework.

**Plug-In Lifecycle**

A plug-in is installed in the communication client by compiling the plug-in code into a shared library (DLL), copying the DLL to the mobile device, and

---

**Figure 2.**
**Plug-in framework architecture.**
adding the name of the implementing plug-in class to the persistent settings of the IMS communication client.

A plug-in can indicate that it depends on another plug-in by providing a reference to its unique identifier. In addition, a plug-in can give itself a relative priority. These two parameters are used to define the initialization order of multiple plug-ins, which is useful, for example, to ensure that the tab order in the main interface remains consistent.

During initialization, the plug-in can add graphical user interface (GUI) elements to the IMS communication client (such as tabs in the main interface); it can start any background threads; and it can start listening for IMS events triggered by the IMS communication client, such as a successful IMS registration or a relevant SIP NOTIFY message.

A common mechanism is provided to store all the settings for the plug-ins in a single location. The plug-in can query for settings, or it can specifically listen for any configuration change events as they occur while the application is running.

At run-time, the plug-in framework provides a loosely coupled mechanism for communicating between plug-ins and launching “actions” between them. The names of the actions are published systemwide and any component that understands the semantics of the input and output parameters for a given function can use it. For example, one plug-in that manages email exports a “ComposeTo” action that pops up a user interface for email composition, taking an email address as a parameter. Anyone can test whether the email plug-in is deployed and profit from its functionality using this action.

A plug-in is not specifically notified when the application is shut down, but it can detect any changes to IMS connectivity and is given a chance to clean up its state.

GUI Integration

The plug-in framework is especially useful for integrating plug-ins seamlessly into the main application user interface including:

- Adding icons or controls to an information banner above the main dialog window, and
- Popping-up standardized notification and busy messages.

Frequently, a plug-in will add a single tab to the main application window when it is first initialized, and the functionality of the plug-in is invoked by the user interface on this tab. The GUI API permits tabs to be added and removed dynamically from the application. Figure 3 shows GUI integration with the plug-in framework.

In general, the IMS communication client permits the user to configure the client and all of its plug-ins using a single dialog launched from the menu bar. For a plug-in to integrate its configuration user interface into this dialog, it uses a GUI toolkit that simplifies the task of displaying, modifying, and saving plug-in configuration information.

There are two settings that are considered to be best practices for a plug-in. The “Active” setting is a Boolean expression that can be used to disable the plug-in without uninstalling it. When this value is false, the plug-in should still be initialized, but it should refrain from adding any tabs, launching any background processes, or listening to IMS events. However, it should still permit the user to configure it, so that it can be reactivated. Another setting called “ExpertSettings” is used to filter advanced or debug functionality from the configuration tab for the plug-in.

Plug-ins can add user controls to the information banner above the main client display. The best practice is to use this area sparingly to show small state icons or notifications for the plug-in.

Finally, the plug-in framework provides some simple, standardized methods to pop up information dialogs. A developer could implement the same functionality without the plug-in framework, but the plug-ins provide a consistent look and feel for a clean integration into the IMS communication client. These methods fall in the category of either notifications (a message interrupting the user, which can be cleared) or a busy message (a message interrupting the user with a busy cursor, indicating that the client is blocked during processing).
Address Book Integration

The IMS communication client provides a powerful dynamic address book (DAB) that contains a list of all the user’s contacts and provides presence information for some of them. The DAB is also the launching point for IMS communications, such as creating a voice call or sending an instant message.

The plug-in framework provides functions for getting information about the contacts and groups found in the DAB. For example, an email plug-in can attempt to auto-complete an address field from the email addresses found in the user’s DAB. The accessible information stored in the DAB includes, for example, the contact’s name, nickname, email address, SIP uniform resource identifier (URI), phone number, and picture.

The plug-in framework permits new features to be integrated in the IMS client so that the user experience of selecting contacts and communication services from the DAB remains consistent across all of the installed plug-ins.

Communication Events

In the IMS network, many communication tasks involve the exchange of multiple SIP signaling messages. The plug-in framework provides a very high level interface that analyzes the state of the terminal in order to provide a simplified way for plug-ins to react to ongoing user communication. For example, instead of listening to the sequence of SIP messages that indicate that a call was suggested and eventually accepted, a plug-in can simply ask to be notified when a voice call starts. This is particularly useful for plug-ins agnostic about the underlying technology of the voice call; a legacy call using Global System for Mobile Communications® (GSM®) is managed using the same events as a SIP Voice over IP call.
The communication events include, for example incoming calls, user-initiated calls, accepted/rejected calls, call termination, and sending or receipt of instant messages or push-to-talk transmissions. The specific parameters of the call or communication are sent to the plug-in.

The user can also use the communication interface to create new calls or instant messages or programmatically accept or reject calls created by other plug-ins.

**SIP Integration**

Although the communication interface provides a high-level mechanism for reacting to ongoing communications with the terminal, some plug-ins need more access to the SIP stack in order to create their customized SIP messages. The IMS communication client contains a powerful and flexible SIP stack implementation. The plug-in framework API provides some access to SIP messages passing through the SIP stack. The SIP interface has been abstracted from the SIP stack implementation in the IMS client, protecting any third party plug-ins from future changes to the SIP stack. The main focus has been to provide access to the SIP event notification system, e.g., SUBSCRIBE, PUBLISH, and NOTIFY messages [2].

The plug-in framework provides an easy mechanism to subscribe to an event package by automatically generating a SUBSCRIBE method. Plug-ins can listen for incoming SIP NOTIFY messages and are responsible for filtering out which ones are relevant to the plug-in function. For example, to implement push email notifications using SIP notifications, the plug-in would subscribe to push email events published by an application server in the IMS network and listen for all incoming notifications, filtering and processing the relevant ones.

In addition, the plug-in can be notified when the terminal has successfully registered and authenticated to the IMS network (via a sequence of low-level SIP messages). Typically, a plug-in would listen for successful IMS connectivity in order to subscribe to its own event package and generate and send PUBLISH messages when required.

If possible, another event is generated just before the terminal is unregistered from the IMS network, and this is the appropriate time to unsubscribe. However, if the mobile device suddenly loses network connectivity, the plug-in will not have an opportunity to unsubscribe cleanly and must clean up its state when the connection is restored.

The SIP event notification framework is also used to deliver presence information; the plug-in framework provides a helpful toolkit for this special use.

**Future Work**

While the plug-in framework offers a set of stable, open interfaces for third parties to develop their IMS features, introducing new plug-ins is an excellent way to identify missing functionality. Using an acceptable versioning strategy so that existing plug-ins continue to work, new and useful features can continue to be added.

Deploying a new plug-in to an existing installation of the IMS communication client is currently the responsibility of the plug-in provider. A toolkit to automate this process would enhance the dynamic benefits of using plug-ins to add new features. Modifying settings to include a new plug-in is not a step that the mobile operator will want to leave to the user.

The Alcatel-Lucent 5440 IMS Windows Mobile Client has been developed for two editions of the Windows Mobile 2005 operating system: the Pocket PC edition and the Smartphone edition. The differences between the two are mainly in the user interface; a tactile screen is optional for a Smartphone. The plug-in framework has currently been developed only for the Pocket PC, but in order to reach a wider range of devices, a parallel plug-in framework is being researched for both versions of Windows Mobile.

One of the current restrictions is that plug-ins are started and stopped with the IMS communication client. Research is now under way to develop plug-ins that can be deployed or removed at any time.

**IMS Kernel**

The IMS kernel is appropriate for applications that require independent subsystems and processes with little or no integration into the IMS communication client. By using the IMS kernel, along with an accompanying toolkit, new or existing mobile applications can take advantage of IMS communication features such as voice, instant messaging, and presence [6]. For example, a third party might deploy a game that
permits Voice over IP chat with other participants, or a stock-ticker program might indicate the online presence of the end user’s broker.

Originally, the communication stacks and protocols were closely tied to a single IMS client. Third parties who wanted to use the SIP functionality for signaling (or RTP for streaming, or MSRP for chat) could only access these protocols as a plug-in inside the process of the running IMS communication client. The IMS kernel separates these useful components into another independent process, so that, for example, the SIP stack has its own lifecycle outside the IMS communication client. The IMS client functionality appears unchanged, but it now relies on the IMS kernel.

The specific functionality of the IMS kernel is very similar to the functions offered by the plug-in framework (without any GUI integration functions, since the IMS kernel is a background service). The high-level communication events (such as call starting/ending events or sending instant messages) and low-level communication events (such as creating specific SIP messages) are nearly exactly the same. The complication with the IMS kernel is that these functions need to be efficiently called across process spaces in the Windows Mobile operating system.

The Windows Mobile operating system offers few methods for interprocess communication. On a fixed desktop, one choice might be the .NET Remoting API, which is used to make remote method calls, but this technology is not available on the .NET Compact Framework found on mobile devices. Microsoft Message Queues (MSMQ) is a high-performance message exchange protocol meant for interprocess communication. It is optionally available on the Compact Framework, but few equipment manufacturers support it. At the lowest level of the Windows Mobile operating system, an application can define custom messages to be pumped out to any listening applications, but this is unwieldy and error-prone. Finally, a higher-level protocol such as sockets could be employed to communicate across local processes, but this also adds performance and complexity issues.

Fortunately, as shown in Figure 4, the IMS kernel can be constructed as a Windows Mobile service,
a model for long-running independent services that can be used to interact with IMS-enabled applications.

**Windows Mobile Service**

The Windows Mobile Service model defines an API for long-running applications that provide functionality to other programs, including third party services. The services are automatically loaded and installed at system boot time and can be programmatically started, stopped, and configured via a common interface. The operating system manages the lifecycle and resources of the services; all of the services can belong to a single process while appearing functionally independent. As a result, low-usage services cause very little overhead, while high-usage services remain efficient [4].

As all Windows Mobile services are, the IMS kernel service is written in C++ and compiled to a library (DLL) installed on the mobile device. The system registry of the mobile terminal contains the configuration for all of the services and includes a three-letter identifier (ID) used to reference the service, in this case, “IMS.” The DLL exports functions to implement the common interface for the service, and all are prefixed with the ID. At system boot time, the IMS kernel is started (via its IMS_Init() method) and its methods are hooked to the services API for interprocess communication.

The service can automatically be connected to a port and stream its data like a socket, or it can be accessed like a file or pipe, e.g., via its IMS_Open() or IMS_Read() methods. These two mechanisms are very powerful for streaming data to and from a service [5]. The IMS kernel service, however, exclusively uses synchronous message calls to the service. System calls to the Windows Mobile Device IOCTL() API method are automatically forwarded to the IMS_IOCTL() method exported by the IMS kernel service.

For every functionality that the IMS kernel provides, a control code is defined along with input and output data structures. In order to simplify development for IMS-enabled third party applications, as noted in Figure 4, client layers were written in C++ and C# to provide methods for each functionality, automatically sending the control code and performing the custom marshaling and unmarshaling of the parameters. For example, if an IMS-enabled application wants to send a SIP SUBSCRIBE message [2] in order to receive SIP NOTIFY events, it makes a call to the IMS kernel client layer, where the correct DeviceIOControl() call is made along with a data buffer containing the parameters, i.e., the unique ID of the calling application, the event type, and other dimensions. In the service process, the IMS kernel parses the buffer, records that the application with the given ID needs to send a SIP SUBSCRIBE message, and wants to receive the corresponding NOTIFY messages. If required, the SIP stack is started and the terminal is registered in IMS. The success of the operation and its return values are relayed back in a data buffer, which is parsed by the client layer and returned to the IMS-enabled application.

**Events From IMS kernel Service**

This DeviceIOControl() mechanism is extremely efficient for processing frequent and small synchronous method calls, that is, calls from the client that enter the service and require light processing to obtain a quick response. However, asynchronous method calls are necessary in order to implement any functionality that requires the IMS kernel to trigger a function in an attached application, such as firing a notification when the terminal has IMS connectivity. Callbacks are not supported between processes in Windows Mobile, so asynchronous method calls (notably events raised from the IMS kernel) and long method calls use a different method to keep the service responsive and efficient.

For example, when the application declares itself to the IMS kernel service, it can indicate its interest in IMS connectivity events. Figure 5 provides a view of asynchronous events from the IMS kernel service:

1. Inside the service process, a structure is created linking the unique ID of the application to an incoming message queue.
2. At the same time, the client layer automatically starts a thread (in the client process) that will quickly take the first message off the top of the message queue (in the service process) and bring it back to the client.
3. If the message queue is empty, the event thread in the client blocks on a named event, a Windows
Mobile structure used in synchronization. The name of the event is unique across all of the instances of IMS-enabled applications. Fortunately, named events share a global namespace in the Windows Mobile operating system.

4. When the SIP event arrives to indicate that IMS registration is successful, the service process can place a structure in the application's message queue.

5. This also triggers the named event for that process.

6. In the client process, the event thread recuperates any messages in its queue.

7. The event thread also determines the type of event that occurred, and

8. It performs the necessary callbacks for the client.

In addition to IMS connectivity changes, the client can declare callbacks for other service-initiated actions such as specific SIP NOTIFY messages, presence change events, or configuration change events. In the client layer, a single event recuperation thread is responsible for routing event data from the service message queue to the appropriate callback. This keeps the time that the application spends in the service process to a minimum and ensures responsiveness across all IMS-enabled applications.

In order to manage numerous IMS-enabled applications using the AMICAL interface, the IMS kernel service maintains application-configurable limits on the message queue for each application. This ensures that an unresponsive or crashed AMICAL application does not consume operating system resources indefinitely.

This mechanism for asynchronous delivery of AMICAL events is also useful for synchronous client...
calls that may require heavy processing. The AMICAL client layer transparently converts the synchronous call into an asynchronous call that waits for an event to signal completion.

**Inter-AMICAL Communication**

Although IMS-enabled applications are normally independent processes, the IMS kernel provides an opportunity to communicate easily between them. For example, a golf scorecard application that tracks the presence of the user’s “golf buddies” and reports on their game events can easily provide its own API for other IMS-enabled applications.

An application using the AMICAL interface can declare a set of methods for export, keyed by a string unique to the application. A second application can call any of these methods by using the method name and the unique ID of the application, providing a parameter to be interpreted by the called application, and receiving the returned result. Likewise, the second application can ask to be notified of any events that the first generates during its operation.

This extremely loose coupling requires the first application to have a well-defined interface (the names of available actions and the semantics of their parameters) that the calling application can use to send and interpret its results. Since the semantics of the parameters and return types are not strongly defined, both applications need to handle error conditions carefully and gracefully. This means of inter-process communication is layered over the IMS kernel service and is much simpler than other mechanisms provided by the Windows Mobile operating system. An application that manages a new service on the IMS network can be integrated into other applications simply and efficiently, without a significant development effort.

**Future Work**

As for the AMICAL plug-in framework, the specific target for the IMS kernel service is the Windows Mobile 2005 Pocket PC operating system, not the Smartphone edition. Future versions of the IMS kernel should support both editions, especially since the effort to port it to Smartphones should be relatively simple, given that the IMS kernel does not offer GUI integration services.

The IMS kernel requires the user’s IMS settings to be centralized and shared among all IMS-enabled applications. Although IMS-enabled applications are free to manage their own specific configuration, there is an opportunity to provide methods that transparently centralize all application configuration through the IMS kernel at little development cost. This provides the opportunity to share configuration information easily between applications and even generate configuration change events.

As in the Plug-in Frame, access to the low-level SIP stack provided by the IMS kernel is focused on SIP event management [2]. Future versions of the IMS kernel can strengthen and develop the SIP framework by adding functionality for more SIP message types (for example, the useful SIP INFO message) in order to compete with other SIP APIs such as Java Specification Request (JSR) 281 [1].

The IMS kernel provides a simplified mechanism for IMS-enabled applications to communicate between their different processes, using predefined action names with well-defined structures for parameters and return values. Currently, any applications taking advantage of this inter-AMICAL communication must know the definition of these exported values beforehand. To truly take advantage of this loosely coupled mechanism, future research should be conducted to determine how applications can advertise their capabilities to other applications.

**Dynamic Application List**

AMICAL plug-ins are tightly integrated into the IMS communication client user interface, typically with a tab for their features and services. Likewise, independent IMS-enabled applications that use the IMS kernel have an opportunity to integrate loosely into the IMS communication client using the dynamic application list (DAL).

The DAL is a tab inside the mobile IMS communication client that provides a launching point for all of the IMS-enabled applications on the mobile terminal. As shown in Figure 6, using the inter-AMICAL functions provided by the IMS kernel, each application can specify a user-friendly name and icon to be displayed in this list.
Furthermore, running IMS-enabled applications can provide a hierarchy of IMS-enabled objects to be shown in the DAL, each with an icon, text, and context menu that can be used to call methods exported by the application. For example, the user could expand the golf scorecard application inside the DAL to display a list of “golf buddies,” showing their presence and any interesting golf statistics. Since each buddy is also a valid IMS contact, the user can use any communication capability in the IMS communication client to make a call, send an email, or an instant message.

The hierarchy of objects in the DAL is entirely specified from within the IMS-enabled application; the DAL is simply a summary screen that permits the user easily to access the application in a flexible and ergonomic way.

**AMICAL in Action**

Before being refined as a third party API, the AMICAL interfaces were originally developed in-house to manage and deliver internal product concepts and research prototypes. AMICAL plug-ins and IMS-enabled applications are widely used to extend the IMS communication client.

**Plug-Ins**

Screen shots of plug-in product concepts are shown in Figure 7. The TouchDial plug-in, shown in Figure 7a, adds one-finger dialing to the IMS communication client and is designed for improving the ergonomics of dialing contacts in the user interface. The Windows Mobile operating system has a separate object model for contact information; the TouchDial plug-in consolidates all of the user’s contacts from all sources. It primarily uses the GUI integration and dynamic address book interfaces provided in the AMICAL plug-in frame.

The correlated ID (CID) plug-in, shown in Figure 7b, searches for additional information about an incoming caller on the Internet and user-specified
directories. It uses the high-level communication functions in the plug-in framework to detect incoming call details (for both legacy GSM and Voice over IP calls), then dynamically constructs a new tab with details about the caller. The user can accept or reject the call from within the CID plug-in.

The predictive IMS interactive messaging (PIIM) plug-in shown in Figure 7c adds email, video, and voicemail with push notification to the IMS communication client. It uses low-level SIP functions to subscribe to an application server that detects new mail on the user’s legacy email server or voice message box and creates SIP notification events with the relevant information. The user is notified via a pop-up when there are new messages, and a single user interface on the “Mail” tab provides a unified view of all inboxes. The PIIM plug-in also integrates with the dynamic address book to decorate the sender’s icon to indicate the type of unread messages from that contact.

The Reachability plug-in, shown in Figure 7d, tracks the user as the mobile device moves in and out of wireless local area network (WLAN) hotspots. The terminal detects network change events and permits the user to specify a context for the any new hotspots, such as the relative cost, bandwidth, and “place type” (also known as “sphere”) of the hotspot. This has two practical benefits: (1) when users enter a previously learned hotspot, they are automatically placed in the previously learned sphere, and (2) the learned sphere is used to implicitly update their presence information so that the users’ work colleagues can only see the presence information if users are in a “work” sphere [3]. This plug-in uses the basic GUI integration and low-level SIP functions of the plug-in frame, but more importantly, it provides a set of useful and generic network functions that could be integrated into future visions of the plug-in framework to provide network heterogeneity functions to any interested plug-in.

Applications

The dynamic mobile desktop (DMD) is a product concept that provides lightweight GUI objects, known as widgets, for mobile terminals. New applications and services, in the form of these widgets, can be provided by the operator, suggested to the user, and deployed on the user’s device with a minimum of installation fuss.
Applications are automatically kept up-to-date via the appropriate notifications pushed to the device, as shown in Figure 8. Before the AMICAL project, DMD used an HTTP server on the user’s device to receive and process notifications. The AMICAL IMS kernel permitted DMD to replace the HTTP server on the device with notifications via the SIP event mechanism [2], integrating the project into the IMS network, thus simplifying the local implementation of the DMD client and improving the battery life of the user’s terminal. DMD is a good example of enabling an existing application with IMS functionality without any heavy reliance on the IMS communication client.

The machine-to-person user integration (M2PUI) concept is similar to DMD in that it provides lightweight, zero-installation access to applications and services, but with a focus on creating a channel to a third party service provider. Existing Internet services can be extended to the mobile device, providing rich and structured interactions over SIP notification messages. For example, a user subscribed to the eBay* service can be pushed a notification when he or she no longer has the highest bid in an auction, providing the opportunity to respond appropriately. Figure 9 provides a view of this notification. Although M2PUI is independent of the IMS communication client, it uses the DAL to give the impression of fully integrated applications.

**DSE: Smart Third Party Mobile Application Management**

The AMICAL plug-in framework and IMS kernel permits third party applications and services to take advantage of IMS capabilities of the user terminal in an efficient and structured way. These third party applications will take advantage of the IMS infrastructure to add useful features to a user’s terminal in unexpected and innovative ways. One of the remaining problems, however, is to deploy and manage a potentially large suite of these applications in a way that does not overwhelm the end user.

In addition, third party applications and services available on mobile terminals must adapt to the context of the user. For example, it is dangerous to show a text message to a user when driving a car. The best solution in that particular case would be to use the voice capabilities of the terminal instead. It becomes even more important when considering that the number of third party applications and services available is continuously growing. Unfortunately, other mobile terminal constraints remain. The screen is usually small and consequently allows the display of

![Figure 8. Screenshot of an AMICAL application.](image)

AMICAL—API for Mobile IMS Client Application
API—Application programming interface
IMS—IP Multimedia Subsystem
IP—Internet Protocol

Windows Mobile and the Windows logo are trademarks of Microsoft Corporation.

![Figure 9. Sample notification message.](image)

eBay and the eBay logo are trademarks of eBay Inc.
Madagascar is a trademark of Dreamworks Animation LLC.
only a few applications at a time. The one most applicable at any given moment may not be always easily accessible.

The dynamic software environment (DSE) product concept proposes a solution to these problems by providing end users with the most pertinent service when needed, as adapted to their context. AMICAL provides a basic framework on which research in this area can be conducted.

Assumptions

DSE has a client component and a server component. The DSE client is installed on end user terminals and the DSE server is on the network, accessible by any DSE clients.

Being integrated into the IMS platform, the DSE client side uses AMICAL APIs to leverage functionalities such as presence, authentication, and multidevice registration, and IMS communication technologies such as Voice over IP, instant messaging, or push-to-talk.

A third party application or a service is represented on the DSE client as a widget directly accessible from the desktop. Although the third party application or service could be available in another form on the end user’s device (as IMS-enabled applications or plug-ins), this last assumption is made to simplify the discussion of the behavior of the dynamic service environment. The DSE client allows for the management of user widget lists, making it possible to add new widgets by browsing and selecting them and removing widgets as required. Finally, DSU provides users with the ability to specify preferences around which kind of widgets they are interested in, making it possible for the DSE server to make intelligent recommendations when new widgets are available.

Organizing the Desktop

Consider the following scenario. John is an average user with a mobile terminal on which a DSE client is installed and running. He has a few widgets already available on his desktop, but too many to be able to see all of them at the same time. As many users do, John uses his mobile device at home and at work, and he naturally has different needs in these two contexts. Many of the widgets he uses are only useful in a specific context; running them unnecessarily consumes the limited resources on his mobile device.

In order to profit better from the widgets installed on his device, John decides to create different “panels” on his mobile device. Panels are a collection of widgets and their respective user interfaces. For example, a panel can be created to host widgets he uses at home, while another can host widgets used exclusively at work. In this case, only one of them should be visible at a time. He distributes his widgets among the panels, choosing which is appropriate in which location. A widget may be placed in more than one panel. For example, his calendar widget is useful in both his home and work location. John can also configure the position of each widget in a particular panel.

Bringing the Desktop Alive

By manually selecting any of his panels, John is now able to access his widgets easily. However, the intelligence of the mobile device moving through the IMS network can help automate this selection; John creates rules to bring his desktop to life, so that the most appropriate widgets are always at his fingertips as his context changes.

Creating custom user rules. There are several ways to create rules in the DSE environment. Simple rules can be parsed from natural language; the rule can be written or spoken, using a speech-to-text mechanism. John can also use a rule editor to create more complex rules, using a comfortable and user-friendly environment to specify when a panel and/or widget should be activated. A rule may possibly be proposed by DSE itself; as the dynamic service environment learns and analyzes a user’s behavior and habits. For example, if usage logs indicate that the same widget is opened at approximately the same time every day (such as John’s consulting his personal calendar at lunch), a rule to open the widget at that time can be suggested. Once the associated rule is accepted, John’s calendar is automatically brought to the foreground during John’s lunch hour, and he does not need to start the widget manually. Once a rule is created, John can simulate it to check that the behavior is as expected. Rules can also be given priorities to help DSW handle conflicts between rules, but if an
irresolvable conflict occurs, DSE brings it to John’s attention to determine which action to take (and potentially resolve future conflicts). John can decide to deactivate and/or reactivate rules at any time.

As a basic example, instead of manually selecting the appropriate panel, John can have it automatically selected and displayed according to his location (“When I am at home, display the home panel,” “When I am at work, display the work panel”). The localization of his device can be determined using IMS presence via the AMICAL interface. He can also define rules to change the position of widgets within a given panel (“When I have a meeting, put the clock widget on top”). He can decide to open, hide, or close a widget (“When I arrive at home, start the TV Guide® widget”). Even the behavior of the widgets and his terminal can be modified depending on his context (“No sound during a meeting” or “Stop polling for stock prices in case of low battery”).

Using predefined rules. As an average user, John has the power to create his rules, but he can also benefit from common and useful rules created by his operator. Using predefined rules is a simple means for a user to benefit from advanced widget deployment, and a way for the mobile operator to have a hand in shaping the widget preferences of the user base.

Service providers can also supply a set of rules that logically apply to their specific widget, since they have an in-depth knowledge of the widget’s lifecycle and behavior. For example, suppose the user’s preferred financial institution has created a currency converter widget. The widget is automatically suggested by the following rules:

- “The currency converter widget should be recommended to any users entering a country where their home currency is not used.”
- “The currency converter widget should be removed from the desktop of any users entering a country using their home currency.”
- “The default conversion should be from the foreign currency of the entered country to the user’s home currency.”

The bank deploys its new widget on DSE server. Each time John (or any DSE user) enters a country where the Euro (his home currency) is not the official currency, the DSE server recommends the currency converter widget to him if he does not already have it. If John accepts, the proper default conversion is automatically selected. When John goes back to his original country, the DSE server suggests removing the currency converter widget to free resources on his terminal. John can either follow the recommendation or keep the widget anyway.

Via the DSE server, an official organization can also provide a set of rules, such as forbidding the use of some specific widget. It could also be used by weather institutes to broadcast alerts to DSE users in case of storms. Even enterprises could potentially provide rules to align widget usage with corporate policies. For example, on entering a new corporate site, the user could receive a recommendation for a widget to book meeting rooms in the workplace, or (given the proper authority) the enterprise could forbid the use of a specific game widget while at work.

Environment Sharing

Thanks to the rules associated with his widgets, John’s desktop is well organized. Depending on his context or location, the panels switch automatically, the order of his widgets in his panels changes, new widgets are automatically recommended, and widgets behave appropriately in his given context. John wastes less time looking for a specific widget, and his mobile device is more responsive to his needs. John is quite happy about it and shows his environment to several of his friends in his IMS address book. He can recommend some of his widgets to them; he can share an entire panel and any of his associated rules. The sharing functionality of the dynamic service environment is a way for innovative, useful widgets to expand their user base and reach an interested target audience via valuable word of mouth.

Conclusion

The AMICAL project has investigated many aspects of third party development on an IMS-capable mobile terminal, providing different levels of integration with two simple and useful toolkits: the AMICAL plug-in framework and the AMICAL IMS kernel.
Third parties can easily add IMS communication functionality (such as presence information, Voice over IP, or chat) to their applications and services, relying on these robust and functional toolkits, ideally delivered with the user’s mobile terminal. In addition, applications and plug-ins can be developed using only the stable, open interfaces published by the AMICAL project, without needing to worry about the evolution of the underlying IMS communication client. Engaging a third party developer community can provide innovative services that add value to the IMS network and the mobile operator.

The end user benefits from this wider range of innovative services, while retaining a consistent and flexible look and feel to the IMS communication client. Using the plug-in framework and dynamic application list, new IMS functions appear completely integrated into the existing and familiar user interface.

Using the plug-in framework, mobile operators can tailor their IMS communication client to the capabilities of the user’s device (for example, only including the push-to-show plug-in on devices that have cameras and high-bandwidth network interfaces). Value-added services and features can be added to the terminal without redeploying the communication client.

Using the IMS kernel, existing applications can be adapted to take advantage of centralized IMS protocols, without having to run in the same process space as the IMS software components. Any new IMS-enabled application can be developed and deployed on the same terminal as an IMS communication client using a modular and independent architecture, yet still appear tightly integrated to the end user.

The AMICAL framework has been the basis for many research concepts and prototypes over the past year. It has recently been refined into an API that is being included as a feature in the Alcatel-Lucent 5440 IMS Windows Mobile Client. This commercialized IMS communication client demonstrates that the underlying IMS client can change dramatically, yet still provide a stable interface for third party developers to work with.

Along with adding value to the user’s terminal and the operator’s IMS network, the AMICAL framework can be used to support further research in future research projects. The dynamic service environment, a large-scale service management and deployment system, uses AMICAL both as a basis for IMS communication to a mobile device and to define IMS-enabled widgets on the mobile terminal.

*Trademarks

eBay and the eBay logo are trademarks of eBay Inc. GSM and Global System for Mobile Communications are registered trademarks of the GSM Association. TV Guide is a trademark of TV Guide Magazine Group, Inc. Windows, Windows Mobile, and the Windows logo are trademarks of Microsoft Corporation.

References


(Manuscript approved April 2008)

RYAN SKRABA is a research engineer with Alcatel-Lucent Bell Labs in Villarceaux, France. He holds a B.Eng. degree with distinction in computer engineering from the University of Victoria in Victoria, Canada. He has worked across a wide range of topics relating to mobile applications and end-to-end services, including mobile architecture, ergonomics, and network heterogeneity.
OLIVIER LE BERRE is a research engineer with Alcatel-Lucent Bell Labs in Villarceaux, France. He obtained a master’s degree with distinction in computer sciences from the University of Brest, France. He has worked across a range of topics relating to mobile phone applications and end-to-end services, including participation in European projects.

PATRICK LEGRAND is a research engineer with Alcatel-Lucent Bell Labs in Villarceaux, France. He obtained a master’s degree in engineering from the Ecole Nationale Supérieure des Télécommunications de Bretagne (ENSTB), in Brest, France. During his tenure at Alcatel-Lucent he has been involved with mobile phone applications and end-to-end services development, and he now coordinates technical studies, projects, and collaborations.

JOHANN DAIGREMONT leads activities on IMS converging applications development at Alcatel-Lucent Bell Labs in Villarceaux, France. He graduated from the University of Besancon, France, with a master’s degree in computer sciences, specializing in parallel and distributed computing. He previously worked on intelligent networks, next-generation networks, and location-based services, before specializing in mobile applications and end-to-end services.◆