Plug-In Architecture for Mobile Devices

Madhu Keshavamurthy
Jung Soo Kim
Mona Li
Vichaya Sagetong

August 2002

Product Line Practice Initiative

Unlimited distribution subject to the copyright.

Technical Note
CMU/SEI-2002-TN-023

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2002 by Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and “No Warranty” statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Contents

List of Figures

List of Tables

Executive Summary

A businessman travels to another city. During the trip, he keeps track of the money he spends using the Expense application on his mobile device. His secretary asks him to send her the updated records of his expenses for the trip so that she can close the accounting report for the month. To do so, he activates the Expense application (which is compliant with the plug-in architecture for mobile devices [PAMD]), and clicks on its Plug-Ins button. The list of plug-ins that appears on his device's screen includes plug-ins for totaling his expenses, emailing and printing his expense records, and transmitting expense information to other mobile devices. There are multiple plug-ins for emailing, each from a different vendor—he chooses the one he prefers and emails his expense records to his secretary.

This real-world scenario explains the aim of PAMD architecture. PAMD enables end users to satisfy their needs regardless of the mobile application's original capabilities. This encourages service providers to provide better services by developing newer plug-ins for the existing applications, enabling them to capture the market. This also enhances competition amongst plug-in developers, which, in turn, creates better, more powerful plug-ins to better serve end users' needs. And customers don't have to spend money on new applications to meets their needs. So as a result of PAMD, customers incur less cost and get better products and service.

PAMD is an architecture that enables this interoperability between applications and enhances the usability of mobile devices.

Abstract

This technical note describes plug-in architecture for mobile devices (PAMD)—an architectural specification that extends the function of applications in mobile devices. Users gain major benefits when the functionality of applications that run on these devices can be extended through the addition of new services that don't require changes to the application itself. PAMD provides interoperability between applications and plug-ins without sacrificing the performance of the mobile devices on which they run. Because existing applications can be made PAMD compliant with little modification, the development time and costs of adding functionality to them can be reduced dramatically. As PAMD bears the burden of communicating with plug-ins, application and plug-in developers can develop their own products independently and easily use each other's products.

This technical note also describes PAMD's interfaces, how applications and plug-ins interact with them, and the advantages of using PAMD. Also included are several scenarios that explain the architecture and how it can be implemented, and suggestions for extensions that enhance it.

1 Introduction

This technical note, which was written by students in the Masters of Software Engineering program at Carnegie Mellon University, describes plug-in architecture for mobile devices (PAMD). PAMD is an architectural specification that extends the functionality of applications in mobile devices.

1.1 Background of and Motivation for Using PAMD

Mobile devices have become very popular due to their compact design and usefulness. People can use them to conveniently organize their day-to-day work. Mobile applications seldom meet users' satisfaction because of their increasing demand for new features. Mobile device end users tend to have individual needs that may go beyond the original capabilities of an application. This situation has compelled the application developers to bundle more functionality in an application to please their end users. This, in turn, puts a tremendous amount of pressure on developers to add new software functionalities to these devices at a high cost for both developers and users. It also creates the need for the application to extend the architecture's functionality by reusing other available services in the mobile device. Mobile device users gain major benefits when an application can extend its functionality by adding new services that don't require changes to the application itself.

The concept of allowing applications to reuse separately developed components is not new. It is integrated into reusable components commonly known as plug-ins. Plug-ins are software programs that extend the capabilities of an application in a specific way. A common example is a browser plug-in for desktop PCs, such as Apple QuickTime, Macromedia's Shockwave, and RealNetworks' RealPlayer. These plug-ins enable users to play audio samples and view videos from within a Web browser. When the user launches the Web browser and tries to play the audio sample or view the video, these plug-ins will be loaded dynamically into the Web browser and executed. An example of a plug-in for mobile platforms is Sony Clie's PictureGear Pocket, which enables users to select and display digital images on Address Book, PhotoStand, or MS Cam. HackMaster for the Palm OS is another good example of a plug-in that helps applications go beyond their original capability by adding new functionality to the existing applications. For Palm devices, several plug-in architectures are created. ThinkDB, an architecture from thinkingBytes Technologies, allows users to add functionality to ThinkDB plug-ins or to create new ThinkDB plug-ins to access relational databases for mobile devices. Unlike other plug-ins for mobile platforms that are created for a specific purpose, PAMD plug-ins can be used by any PAMD-compliant application. This assembly of services satisfies end users without imposing any cost burden on them.

Similar service-coordination architectures, such as Microsoft's COM, are also implemented on desktop PCs. This type of architecture is becoming more popular in the distributed computing

realm in the form of Microsoft's .NET and Sun's JINI architectures. PAMD provides benefits that are similar to these architectures, but is unique in that it is designed specifically for mobile device platforms such as the Palm OS or Windows CE. On those platforms, the needs and environments of the end user tend to change unpredictably.

PAMD enables service providers to independently develop components that can provide specific services in the form of a PAMD plug-in (henceforth referred to as *plug-in*) that can be used by any application in a mobile device. PAMD also enables application developers to independently develop a PAMD application (henceforth referred to as *application*) to make use of available plug-ins without knowing any details about their capability.

1.2 Objectives of PAMD

The main objective of PAMD is to achieve interoperability between applications and plug-ins on mobile devices. PAMD benefits developers by reducing development time and costs, and benefits end users by dramatically increasing the functionalities that mobile applications can provide. With PAMD, different stakeholders such as application developers and plug-in providers can develop their own products independently and easily use each other's products.

Through the use of plug-ins, PAMD can dramatically increase the functionality that mobile applications provide to end users, regardless of the applications' limitations. PAMD also encourages without restriction a competing market for developing the best plug-ins possible. End users will benefit from that competing market when better, more powerful plug-ins that meet their needs are created.

1.3 Roadmap and Intended Audience of This Document

This document elaborates on the following aspects of PAMD:

- Section 2 lists the architectural requirements of PAMD and the major quality requirements considered for this architecture.
- Section 3 describes the high-level view of PAMD.
- Section 4 describes the PAMD service broker.
- Section 5 describes the PAMD service.
- Section 6 describes the architectural alternatives to the current architecture.
- Section 7 suggests possible future extensions and enhancements for the current architecture.

The intended audiences for this document include

- those interested in service-coordination architecture, an architecture that enables applications to use services without specifying them
- application developers who want their applications to use services
- plug-in developers who want to develop plug-ins to provide services to other applications

2 Architectural Requirements of PAMD

The architectural requirements of PAMD are listed below:

- The main business need for PAMD is to enable the end users to use the existing applications with a new array of services, thus adding value to the mobile devices. The architecture should provide a mechanism for applications to utilize all services, thus providing interoperability amongst applications.
- As different applications interact with each other and with different services, the architecture should handle the error conditions gracefully and prompt the user about the possible error. As PAMD becomes the intermediary between different applications and services, it should be reliable.
- As mobile devices are ubiquitous, it is essential to have an architecture that can be adapted easily to different platforms and different hardware devices. This will reduce the impact on the existing applications. It is also easier for the developers to follow a standard architecture. Thus, PAMD needs to be modifiable and portable so that it can be used on different platforms and devices.
- As mobile devices are performance sensitive, the introduction of a new intermediary should not reduce the overall performance of the mobile device. Slowing down the existing applications will be a heavy price to pay for mobile device users. Therefore, the PAMD architecture should minimize the impact on the performance of mobile device applications.
- PAMD shall be extensible. Enhancements are inevitable for any architecture to incorporate new ideas and improve existing ideas. PAMD architecture should make it easy to add more functionality without affecting the overall architecture.

3 Overall Architecture of PAMD

The high-level structure of PAMD is composed of three layers, each containing different types of executable software components (ESCs). Defined PAMD interfaces facilitate the sequence of interactions among these layers.

In this section we first describe those layers and the ESCs that each one contains. Next we describe the PAMD interfaces that are defined. Finally, we illustrate the sequence of interactions among the ESCs and the underlying platform, and the sequence of interactions in a real-world scenario.

3.1 Layers and ESCs

The three layers of PAMD are client, service broker, and service. The service broker acts as an intermediary between a client and a service. It receives requests from a client and tries to find the right service to satisfy the client's requests.

The client and service layers contain different types of ESCs:

- In the client layer, ESCs are entities that are executed directly by an actor.
- In the service layer, ESCs are entities that are executed indirectly through the service broker.

Figure 1 uses an Acme[1] diagram to show the relationship among the three layers.[2] The figure also shows how the client and service layers connect to each other loosely. Notice that the client layer only needs to interact with the service broker through two interfaces—it does not need to know the identity of any of the entities in the service layer. The layered style dissolves the coupling between the client and service layers. In addition, the internal operation and all the details of the service broker and the entities in the service layer are transparent to the client. This makes it easy for the ESCs to conform to the PAMD specification.

In some cases such as when different services are chained together (a process of passing the output of one service as an input to another service to achieve some specific outcome), the service layer can also be seen as the client layer and can make direct calls to the service broker. This feature enables a specific application to use different services to achieve the desired outcome. This kind of setup resembles the "pipe and filter" architectural style in which

1 Acme is a simple, generic software architecture description language (ADL) that can be used as a common interchange format for architecture design tools and/or as a foundation for developing new architectural design and analysis tools. For further information, go to <http://www.cs.cmu.edu/~acme>.

2 For the textual description of the Acme diagram, see Appendix A on page 27.

one service passes some information to another service and so on, until the desired outcome is achieved.

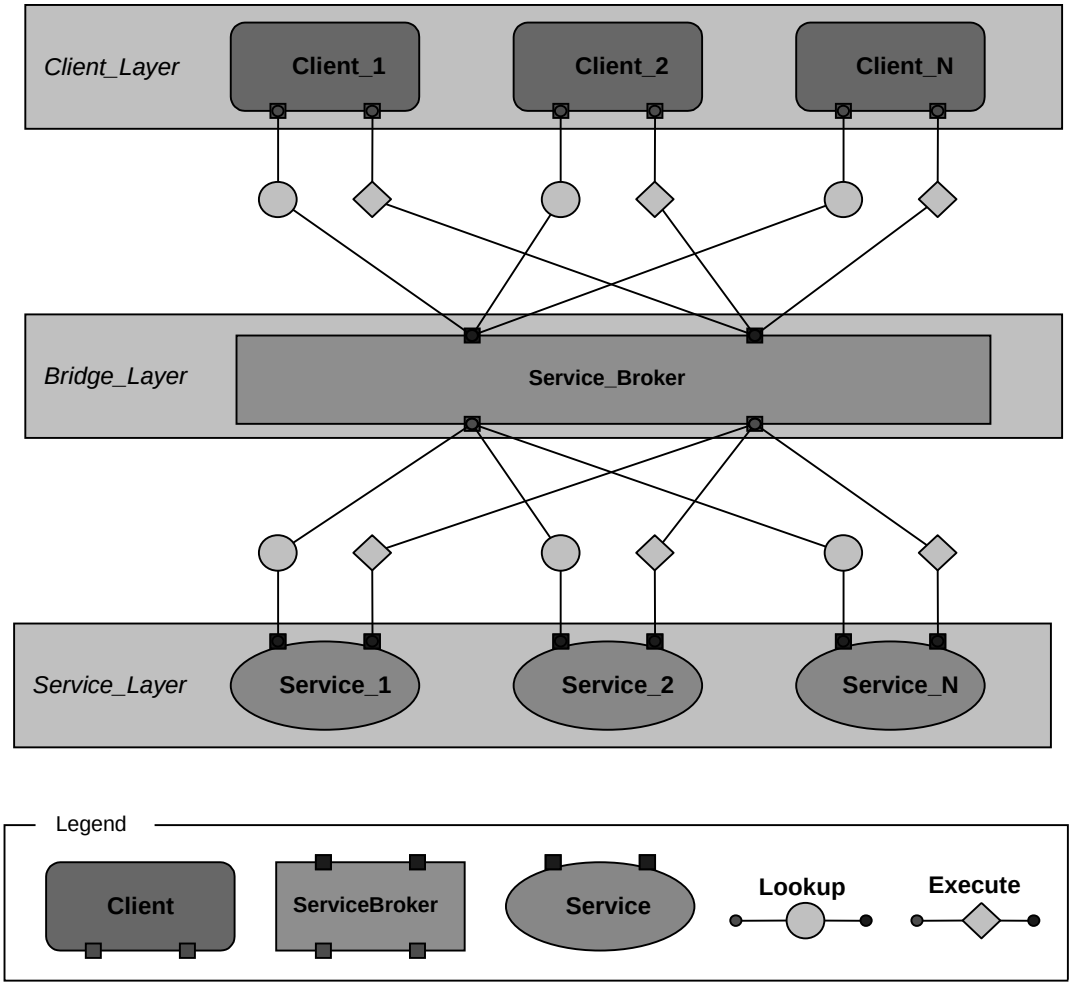


Figure 1: Relationship Among the Three Layers of PAMD

3.2 Assumptions to Underlying Platforms

PAMD makes the following assumptions about underlying platforms. These assumptions should be considered when implementing a PAMD-enabled system on a specific platform.

- The platform should support multiple types of executable entities so that the service broker and plug-ins are distinguishable from applications.
- The platform should enable mobile device users to install and uninstall the service brokers and plug-ins. Plug-ins may need to be updated to newer versions.
- The platform should enable mobile device users to start and stop the service broker. (Similar control over plug-ins is not necessary.)
- The platform enables different executable entities to communicate so that applications use the service broker and the service broker uses plug-ins.

- The communication for executable entities can be bidirectional so that they can receive the results of requests they make to other executable entities.
- The platform manages the list of plug-ins installed in a device and provides it to the service broker when requested.
- The platform provides a means for the service broker to identify each plug-in uniquely.
- The platform doesn't restrict the behavior of plug-ins. In special cases, one plug-in can use another plug-in through the service broker.

3.3 Interface Definition

The interaction with the underlying platform is hidden inside the service broker. To use a service, the client only needs to connect to the broker through two interfaces—each of which is a request to the service broker: one asks for a list of available services, and the other asks for a particular service to be executed.

3.3.1 INTERFACE: PAMDLookupService

```
INPUT 1: A package containing services which plug-ins need to satisfy
INPUT 2: An access key to specific whether the service to be looked
up is a private service or a public service
OUTPUT: A list of service names that supports the given data types
```

PAMDLookupService is an interface accessed through the Service Broker that is used to look up a specific set of services (plug-ins). This interface queries the access key and data type set of all the services in a device. A particular service appears in the list only if both its access key and data type sets match respectively.

Along with the lookup request, the client must specify a data type set package that describes the type of service to be executed and the access key for accessing the service. A data type set package describes the capabilities of services that the client can execute. This data type set consists of 1) an input data type that describes the input data provided by the client and 2) an output data type that describes the type of output the client can handle. The input data type is consumed by the service to produce an output that complies with the output data type. A data type explains how to understand the given data block. For example, standard Multipurpose Internet Mail Extension (MIME) types (such as `text/plain` or `image/jpeg`), or file extensions (such as `txt` or `jpg`) could be used for the data type. There must be at least one data type block in the data type set package.

Each service provides a data type set package that describes the capabilities or type of data that the specific service can accept or deliver. This package can contain multiple data type blocks; each consisting of 1) a current type or an input data type that the service can accept, and 2) a target type or an output data type that it must deliver using the input.

Each service may optionally specify its access key, which every application must use in order to look it up or execute it. The service that is created for internal use in a particular client must have a specific access key for that client to access it and to prevent other clients from using it

without the correct access key. The client that wants to lookup or execute the internal service must provide the correct access key to be eligible to access the service.

Lookup protocol can find the right service only when the requested data type set contains a subset of the data types that a service can handle. Matchmaking is a process of matching the combination of either "input/no input" with "output/no output," which gives four possibilities. Figure 2 illustrates the matchmaking process of the lookup service.

In Figure 2, service W can

- consume data type `text/plain` and produce `image/jpeg` as a result
- handle data type `audio/midi` without generating any output
- consume nothing and send `text/plain` back to a client. In this case, client A can use service W because that client wants to send `text/plain` and get `image/jpeg` back. It also wants to send `audio/midi` without receiving any output from the service. On the other hand, client A cannot use service Y because that service cannot handle the combination of input `text/plain` and output `image/jpeg`. Client B can use Service W because it wants only the combination of no input and output `text/plain`, a combination that service W can handle.

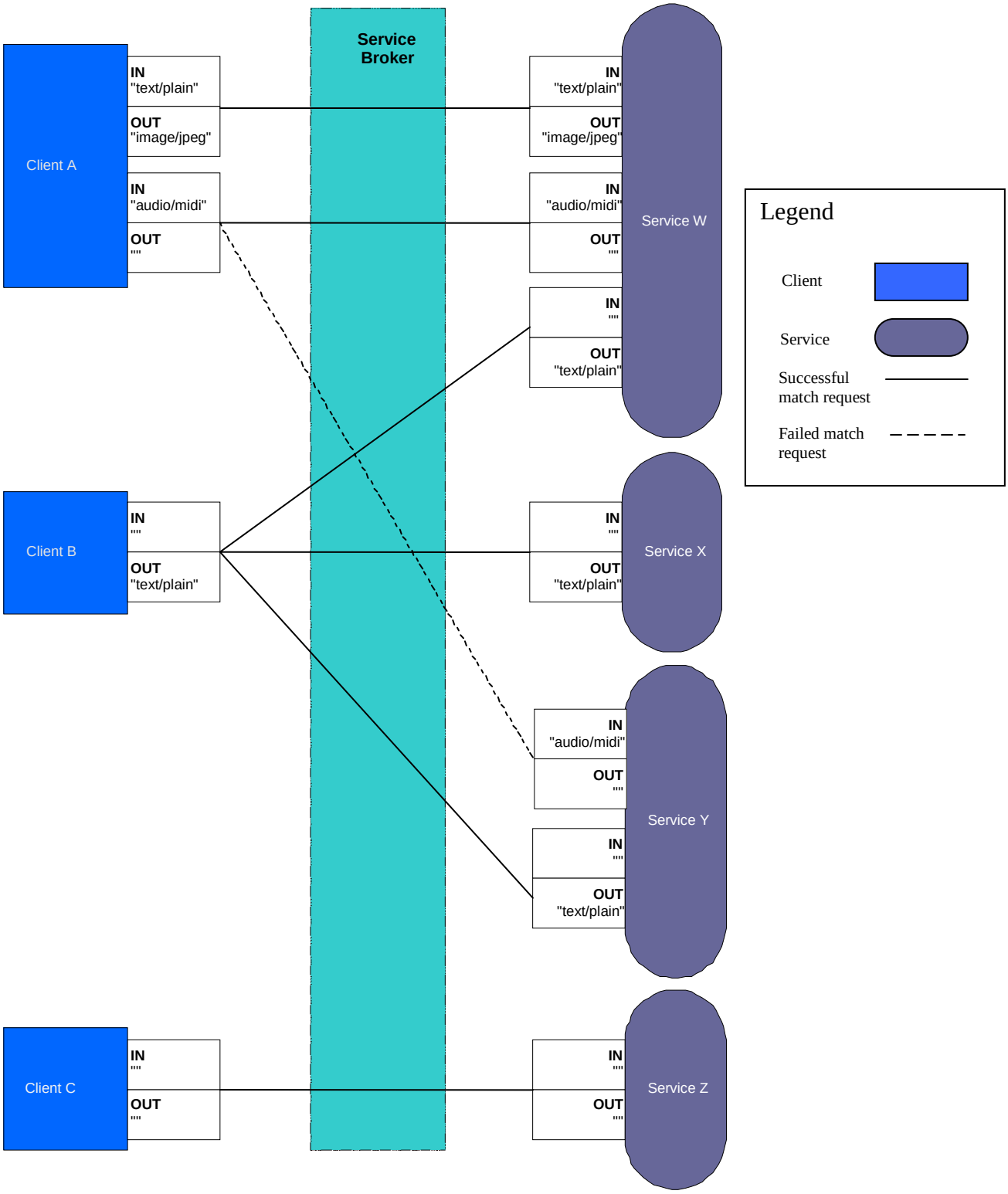


Figure 2: Matching of Client and Service During Lookup

3.3.2 INTERFACE: PAMDExecuteService

```
INPUT 1: Service identifier
INPUT 2: Input data package (if any) for service to consume
INPUT 3: An access key to specify whether the service to be executed is
a private service or a general service
OUTPUT:  Output data package (if any) from service to client
```

PAMDExecuteService is an interface accessed through the Service Broker to execute a selected service. A client must send an input package and an access key to execute a specific service. The input package includes multiple input data blocks. After being executed, the service sends back an output package if required. The output package can contain multiple output blocks in response to each input block.

The package is the unit of communication between clients and services. A package is used as a container to store data blocks. Each data block contains the data; a data type that explains how to interpret the data's contents; the status of the data block, which specifies whether to destroy the data block after deleting the package; and a target data type that describes the data type of the expected result.

Each data block in a package contains the input data type that a service can handle and an output data type that the service will produce by consuming that input. A data block is a pair of data type sets, and a package is a set of data blocks. Each data block in the input package must be valid. For example, take an input package that contains the three data blocks listed in Table 1.

Current Type	Target Type
text/plain	image/jpeg
text/plain	""
image/jpeg	text/plain

Table 1: Data Blocks in Example Input Package

A service would have to generate an output package that contains two data blocks: one with a current type of `image/jpeg` and a target type of `""` to respond to the first input data block; and the other with a current type of `text/plain` and a target type of `""` to respond to the last input data block.

3.4 Sequence of Events

To further explain the relationship among the PAMD layers, Figure 3 illustrates the sequence of interactions among the ESCs and the underlying platform.

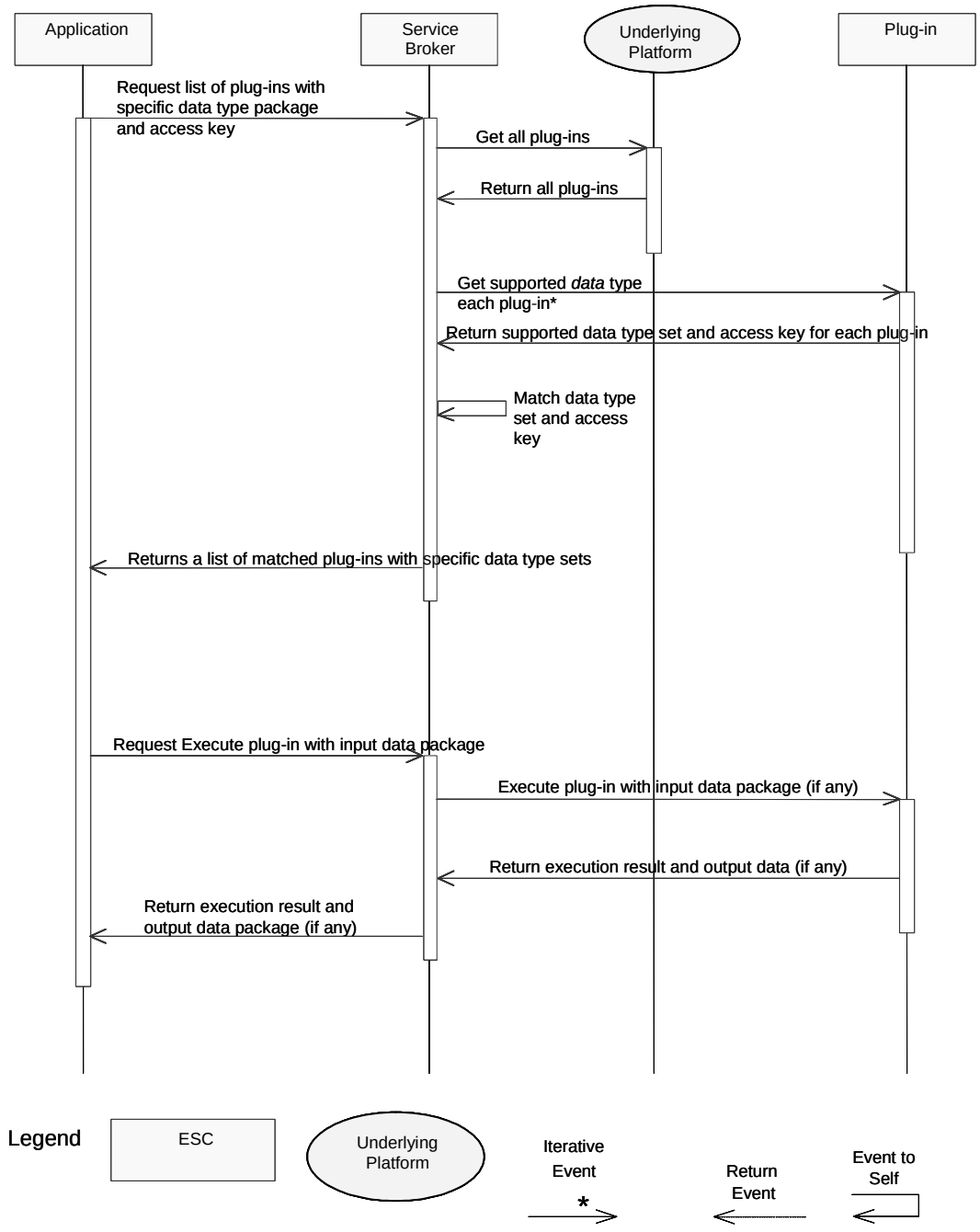


Figure 3: Sequence of Events Among ESCs and the Underlying Platform

Consider this real-world scenario:

> *A businessman travels to another city. During the trip, he keeps track of the money he spends using the Expense application on his mobile device. His secretary asks him to send her the updated records of his expenses for the trip so that she can close the accounting report for the month. To do so, he activates the Expense application (which is compliant with the plug-in architecture for mobile devices [PAMD]), and clicks on its Plug-Ins button. The list of plug-ins that appears on his device's screen includes plug-ins for totaling his expenses, emailing and printing his expense records, and transmitting expense information to other mobile devices. There are multiple plug-ins for emailing, each from a different vendor—he chooses the one he prefers and emails his expense records to his secretary.*

In terms of the interaction diagram, the interaction starts when the businessman clicks on the Plug-Ins button in the Expense application. Once the button is pushed, the application sends a request to the service broker with the access key and MIME-type `text/plain`, which describes the type of input data that it wants the plug-in to accept and directs the plug-in to produce no output. The service broker then contacts the underlying platform of the businessman's mobile device to get all the plug-ins in the system. Then the service broker asks each plug-in for its servable data type set and its access key. It sends back the names of plug-ins that can serve MIME-type `text/plain` and produces no output and the access key. Now the businessman sees the list of plug-ins sent from the service broker on the Expense application's screen. After he selects one email plug-in, that plug-in's name and the expense records are sent to the service broker. The broker, in turn, passes that information package to the email plug-in. The plug-in emails the records and displays a confirmation about the delivery.

4 Service Broker

The service broker serves as a bridge between the client and service layers. In this section, we describe its role, internal architecture, and other feasible alternatives. We also discuss the relationship and interaction between the software components inside the service broker and the sequence of their interactions in the real-world scenario.

4.1 Internal Architecture

The service broker lies between the clients and the services. It encapsulates the interactions between the clients and the underlying platform and services, and provides two interfaces to the clients: `PAMDLookupService` and `PAMDExecutiveService`. The `PAMDLookupService` interface looks up all the available services for a given data type set; the `PAMDExecutiveService` interface executes a particular service and passes the result, if any, to clients.

With this approach, the details and existences of services are hidden from the clients. This simplifies the concern of clients and makes all services transparent to clients, thus decreasing the cohesion among components.

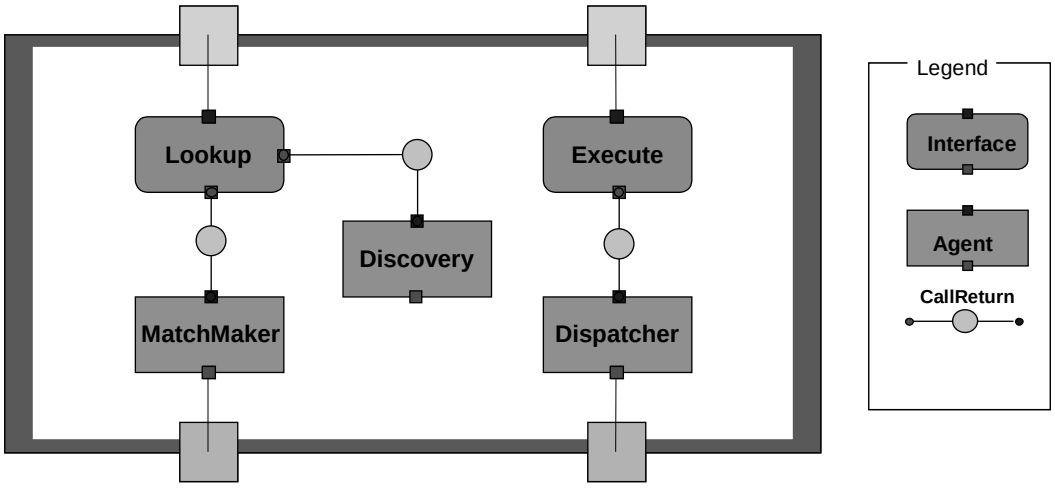


Figure 4: Internal Architecture Diagram of the Service Broker

The above diagram shows the internal architecture of the service broker. It consists of five internal software components (ISCs). Two interface ISCs serve as the front-end of the service broker and interact directly with the clients. Three agent ISCs perform the core tasks inside the service broker.

Each ISC is described briefly below:

- lookup interface ISC—accepts service lookup requests with the desired data type set and access key from the clients and returns a list of available services that satisfy the given request
- execution interface ISC—accepts service execution requests with or without data from the clients along with the access key and returns the result of the execution, usually with transformed data
- discovery agent ISC—discovers all the existing services in a mobile device
- matchmaker agent ISC—compares a given data type set and access key to the supported data type set and access key of each service, and produces a list of services that can handle them
- dispatcher agent ISC—executes a particular service. It checks the validity of the input data and access key before sending a request to a service.

Note that only the interface ISCs can interact with the clients, and only the agent ISCs can talk to services. This implies that the service broker itself has a layered structure.

4.2 Interaction Among Internal Software Components

The following diagram explains how ISCs interact for each service request. Two interface ISCs trigger all the subsequent actions. The lookup service request always precedes the execute service request. The information retrieved in the lookup service will be used when serving the execute service request.

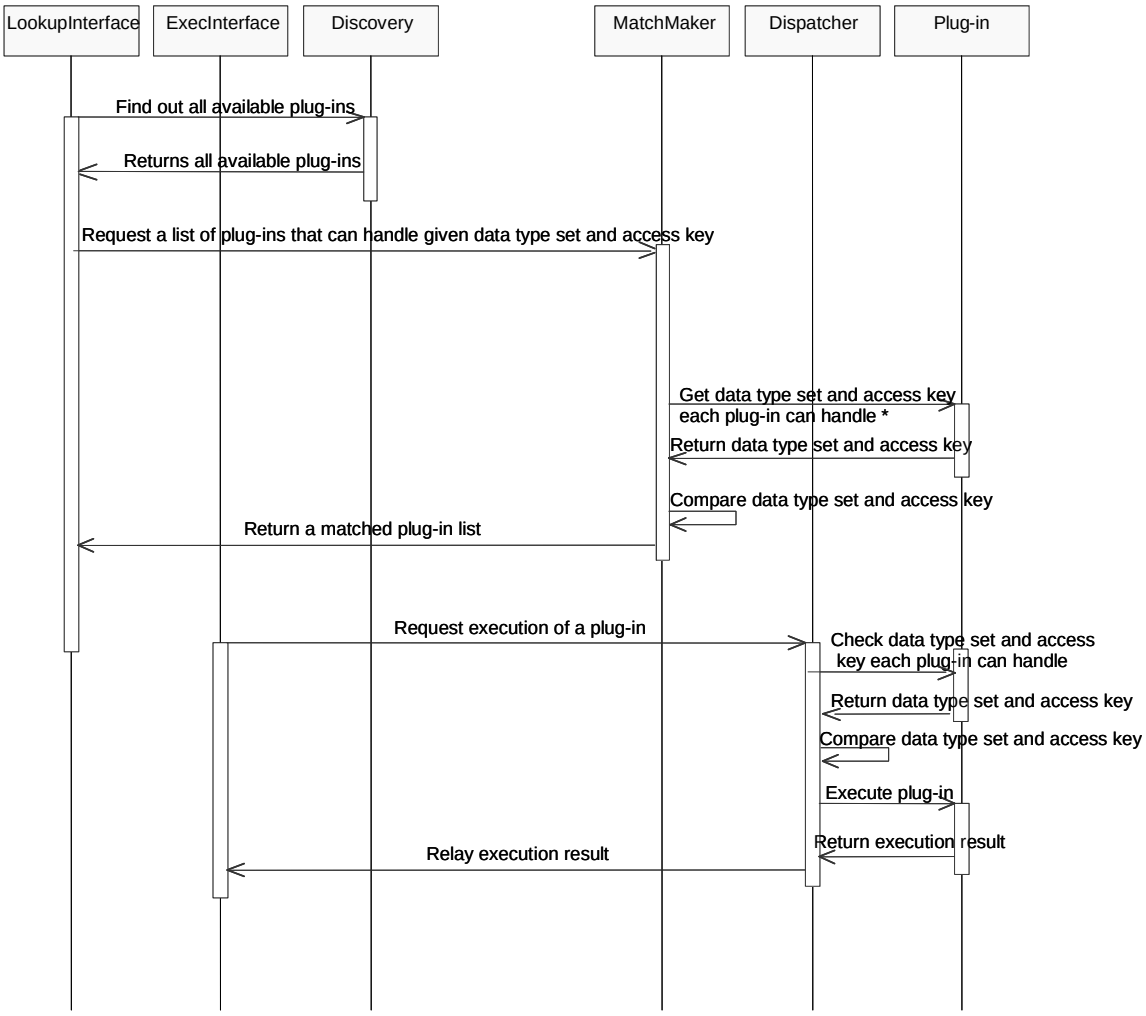


Figure 5: Internal Interaction Diagram of the Service Broker

Considering the real-world scenario presented on page 11 in terms of the interaction diagram, the interaction of this internal architecture starts when the businessman clicks on the Plug-Ins button in the Expense application, and the service broker, specifically the LookupInterface component, receives a request along with MIME-type `text/plain` as the input data type and output type as none. Inside the service broker, the LookupInterface component requests the Discovery component to find all the available plug-ins. Next, the LookupService component requests the MatchMaker component to find plug-ins that can serve MIME-type `text/plain` and the access key. MatchMaker uses the plug-in information to contact each plug-in. It matches the data type set and access key of the incoming request with that of plug-ins and returns a list of matching plug-ins to the LookupService component. This component then passes the list back to the Expense application that, in turn, shows it to the businessman.

The interaction resumes when the businessman chooses one email plug-in, and its name and the expense records are sent to the service broker, specifically to the ExecInterface component. From there, the following occurs:

1. The ExecInterface component requests the Dispatcher component to execute the requested email plug-in with the data package and access key.
2. The Dispatcher component makes a sanity check of the plug-in to be executed with the data type set and access key.
3. The plug-in then emails the expense information to the businessman's secretary and displays a confirmation message in the output data package.

5 PAMD Service

In this section, we discuss the internal architecture of a PAMD service, a service that is provided typically by a plug-in. Recall that a service is an entity that is executed indirectly through the service broker. This section describes the architecture within the PAMD service and the interactions among the service's components.

5.1 Architectural Diagram

The architectural diagram of a service is shown in Figure 6. The square boxes along the top border represent the lookup service and the execute service for the service broker, respectively. Each is bounded to the interface components.

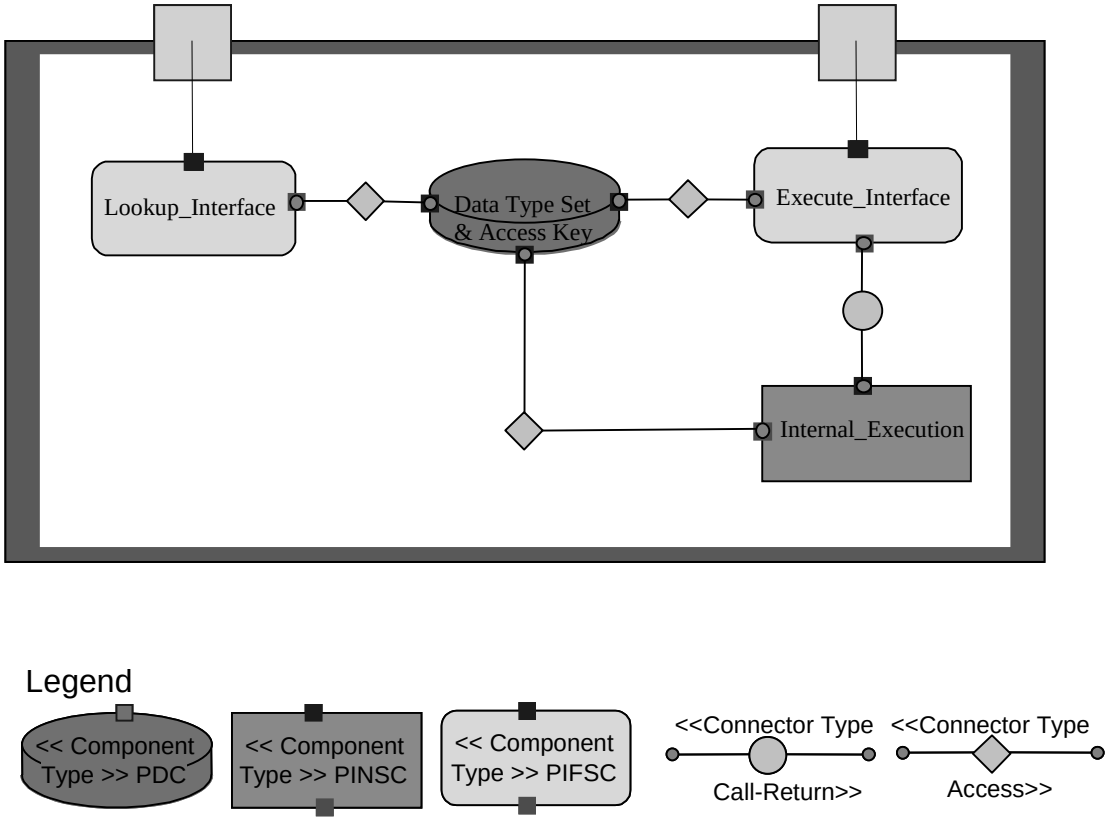


Figure 6: Internal Architecture Diagram of the PAMD Service

5.2 Component Interactions

There are three types of components: the Service Interface Software component (SISC), the Service Data component (SDC), and the Service Execution Software component (SESC). Each is described below.

- Service Interface Software Component (SISC)—represents the front-end of a service that interacts with the service broker. There are two SISC components:
  - Lookup Interface component—receives a request from the service broker for service information. It retrieves the data type set and access key from the Data Type Set and Access Key component (described below) and returns them to the service broker.
  - Execute Interface component—receives a request from the service broker to execute the service. It passes the input data package (if any) to the Internal Execution component and returns the output data package (if any) back to the service broker. This component can access the Data Type Set and access key component to check the accuracy of the input data types before calling the Internal Execution component.
- Service Data Component (SDC)—stores the service information. There is one SDC component:
  - Data Type Set component—stores the data types that the service can handle. The data types can be retrieved by the Lookup Interface component and sent to the service. Before executing a service, the Execute Interface and Internal Execution components check the data type of the inputs through this component.
- Service Execution Software Component (SESC)—performs the specific service. There is one SESC component:
  - Internal Execution component—takes the input data (if any) and performs the appropriate tasks. It generates output and then passes it back to the Execute Interface component.

6 Architectural Alternatives

In this section, we discuss the four major architectural decisions made before the final architecture design. Then we analyze the pros and cons of architectural alternatives to each decision.

The service broker discovers available services whenever there is a lookup request, rather than permanently managing them. Some of the issues considered for the current Service Broker architecture are explained below:

- It might be technically difficult (or even impossible) to keep track of the available plug-ins in a mobile device since the service broker may not know when services are installed or removed.
- Though it is possible, letting all services maintain identical functionality for the registration may waste precious RAM space in mobile devices. Having a separate component to deal with this problem would help to save considerable space as the services get rich. The amount of space needed would remain unchanged even if the number of services in the mobile devices increases.
- Reducing the work needed for each service, even by a small amount, will save a lot of work when several services coexist. This reduction will eventually promote the popularity of PAMD in the industry.

6.1 Plug-In Execution: Indirect Vs. Direct

In the current architecture, applications request that the service broker executes a plug-in. One of the architectural alternatives considered was to provide applications with enough information about existing plug-ins and let applications execute the plug-in directly. Figure 7 shows how a plug-in is executed in the current and alternative architectures respectively. Note that this alternative reduces the amount of interface in overall architecture. A single direct execution connector between applications and plug-ins can replace two execution connectors between applications and plug-ins passing through the service broker.

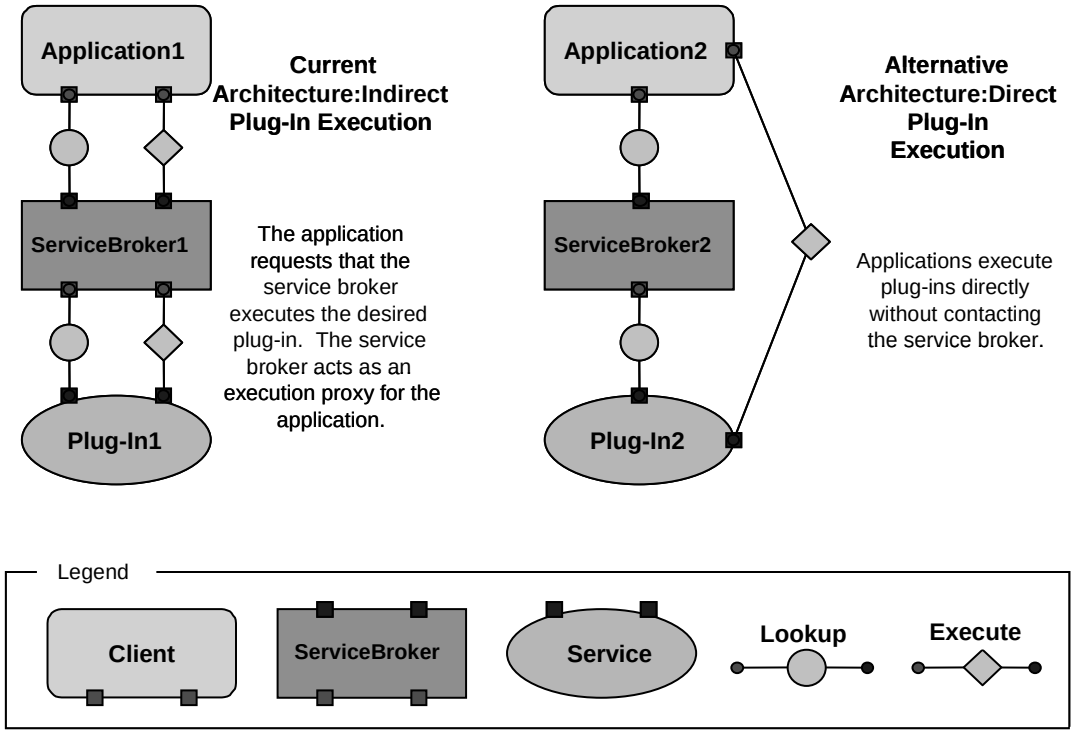


Figure 7: Current Architecture Vs. an Alternative with Direct Execution of Plug-In

This alternative reduces the amount of information stored in the service broker. Even though the alternative architecture will improve performance during plug-in execution, the current architecture was chosen—the reliability, portability, and modifiability of applications are more important than performance. In the current architecture, the service broker does error checking when failures occur during plug-in execution. Doing so promotes reliability. In addition, if a different platform is used, the service broker hides the changes to the platform application program interface (API) that the application uses when executing the plug-in. This makes the architecture more portable. Since the current architecture supports a single point of contact (i.e., the application does not need to connect to the plug-in directly), it reduces the concerns of application developers. Thus, the current architecture supports the modifiability of the application.

6.2 Plug-In Identity: Discovery Vs. Registration

In the current architecture, the service broker discovers the identity of existing plug-ins. One of the architectural alternatives considered was to provide an interface for plug-ins to register their identities when they are added to or removed from a mobile device. Figure 8 shows how to determine the identity of a plug-in in both the current and alternative architectures. Note that the plug-ins actively register themselves to the service broker in this alternative, rather than being discovered.

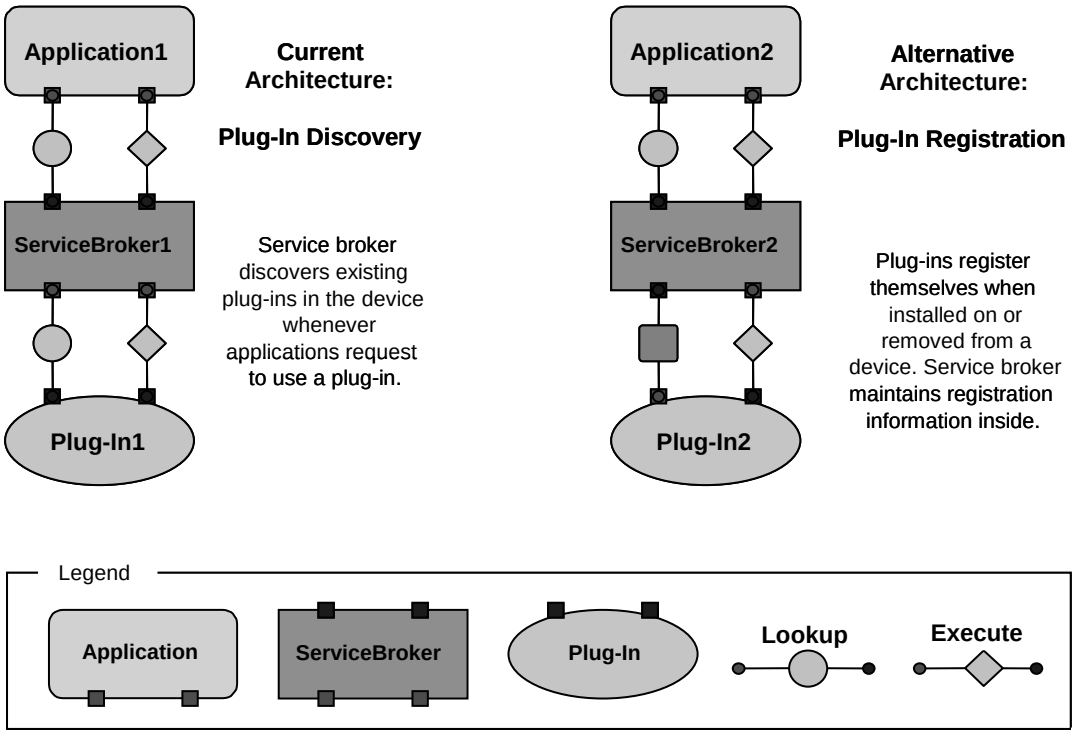
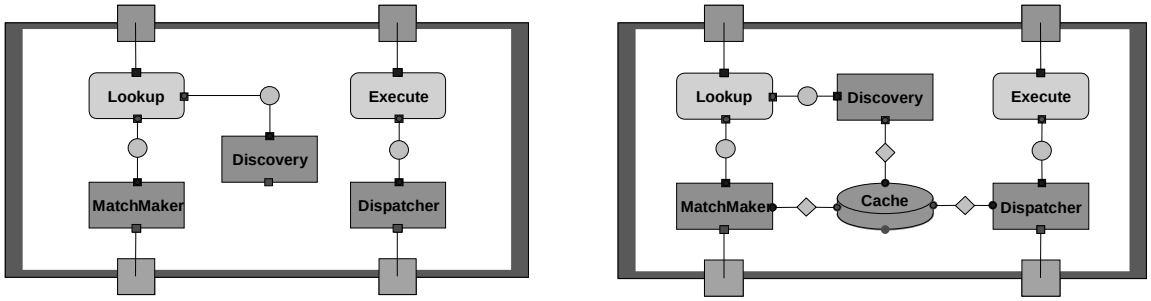


Figure 8: Current Architecture Vs. an Alternative with Plug-In Registration

Assuming that the number of times the plug-in is used is more than the number of times plug-ins are installed and removed, this architecture improves performance during service execution. Even though the alternative architecture will improve performance, the current architecture was chosen—the reliability and modifiability of applications are more important than performance. In the current architecture, the service broker does not have to keep track of plug-in registration. Therefore, the lookup mechanism is less error-prone. If there is an error or a platform upgrade in the development environment, plug-in developers don't need to modify their code because of registration mechanism changes. Also, since lookup is the responsibility of the service broker, the amount of code required to look up plug-ins remains constant regardless of the number of plug-ins. As the number of plug-ins increases, more storage space is saved.

6.3 Information Reuse: Refresh Vs. Cache

In the current architecture, the identity of the specified plug-in is discovered again whenever there is an execute service request. One of the architectural alternatives considered was to store information about the plug-in's identity in the internal cache of the service broker. This information could then be used for executing services. Figure 9 shows how the Dispatcher component acquires the plug-in identity in the both current and alternative architectures. Note that interaction with the underlying platform is excluded from the diagrams.



Component
Current_Service_Broker
(Refresh_Continuously)

Component
Alternate_Service_Broker
(Internal_Cache)

Figure 9: Current Architecture Vs. an Alternative with the Internal Cache

After a performance test, it was determined that there was no performance gain by using cache because accessing the plug-ins using Palm APIs was as efficient as having a cache. Cache was also occupying more memory, as it has to maintain the information about the list of plug-ins. Considering the memory limitation factor in mobile devices, this was expensive. There was no added benefit in using cache when compared to getting the plug-in information each time using the continuous approach, so the choice was made to discover the plug-ins and match them all at once. Plug-ins could be added to or removed from the device at any time. It is essential to procure the updated list of plug-ins, which adds maintenance overhead for the cache. Cache also opens up the architecture for more errors because the latest state of the device must be maintained in the cache. All these factors led us to choose the "refresh continuously" approach.

6.4 Service Broker Internal Architecture: Layered Vs. Monolithic

In the current architecture, interface logic and internal logic are separated into two different layers in the service broker. One of the architectural alternatives considered was to merge them together to reduce the number of components inside the service broker. Figure 10 shows how internal components of the service broker can be combined. Note that each external interface has an associated monolithic component. The Discovery and MatchMaker components are merged together.

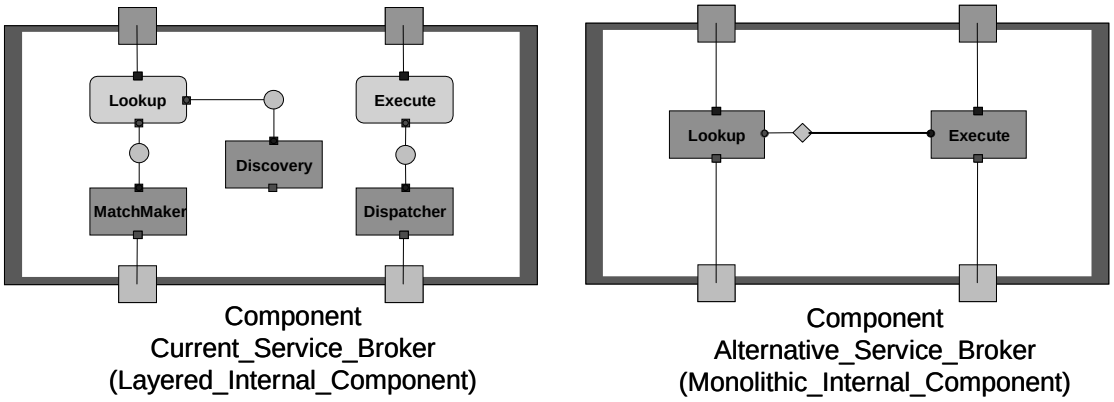


Figure 10: Current Architecture Vs. an Alternative with Monolithic Components

Since there are fewer components and connectors involved in the alternate architecture, the processing time will be reduced a bit. Even though the alternative architecture will improve performance, the current architecture was chosen—the modifiability and portability of applications is more important than performance. When there are changes in the underlying platform of the current architecture, the interface components can simply be modified to deal with them. When this architecture must be implemented on a new platform, the code of the Discovery, MatchMaker, and Dispatcher components can be reused, thus decreasing the time required.

7 Future Extensions

In this section, we show how PAMD can be extended to support distributed operation and how this extension affects existing systems, developers, and users.

7.1 Background

As stated earlier, PAMD architecture is designed for use in a single mobile device. Currently, mobile device users are able to use only services that are available on their local device. Whenever they need a new service, they must install it on the device in the form of a plug-in. The installation process is burdensome for people who only want to use the services for a short time. Additionally, when their devices don't have enough storage to maintain many plug-ins, users often waste a lot of time installing and removing them.

The current trend in mobile computing moves toward the Internet world where abundant information and services are available and continuously updated. People want to use those services and be able to locate them quickly. As a result, people have an increasing need to use multiple services that are distributed over many different machines.

7.2 Overall Architecture of Distributed PAMD

The main goals of distributed PAMD (D-PAMD) are

- Applications should be able to use plug-ins from different devices.
- Plug-ins should be able to serve applications in different devices.
- Different devices with different platforms should be able to work together.
- Any architectural changes should be transparent to existing applications and plug-ins.

To achieve these goals, the service broker needs to be changed to allow the applications, plug-ins, and platforms to cooperate so that they appear to be a single virtual service broker from the application's perspective. This might involve peer-to-peer or group cooperation, depending on the given interconnection mechanism; in this document, the focus is on cooperation between a group of service brokers that share their services.

The following diagrams show an overall architecture diagram of distributed PAMD. Since multiple service brokers are involved, information about their identities should be stored in a central place that could be accessed by all the service brokers. A new component was introduced: Federation Hub that manages the list of connected service brokers and is similar to the Lookup Service (LUS) in the JINI architecture. Service brokers register themselves in a Federation Hub and form a federation of services. Whenever there is a service request from an application, service brokers pass it to the Federation Hub that, in turn, spreads it to other service brokers. After looking up the information about the available plug-ins, the results are

passed back to the Federation Hub. Eventually, the application will get all the available plug-ins in a federation. Though service brokers form a great federate of services, applications and plug-ins don't notice this at all.

Figure 11 illustrates the architecture of D-PAMD.

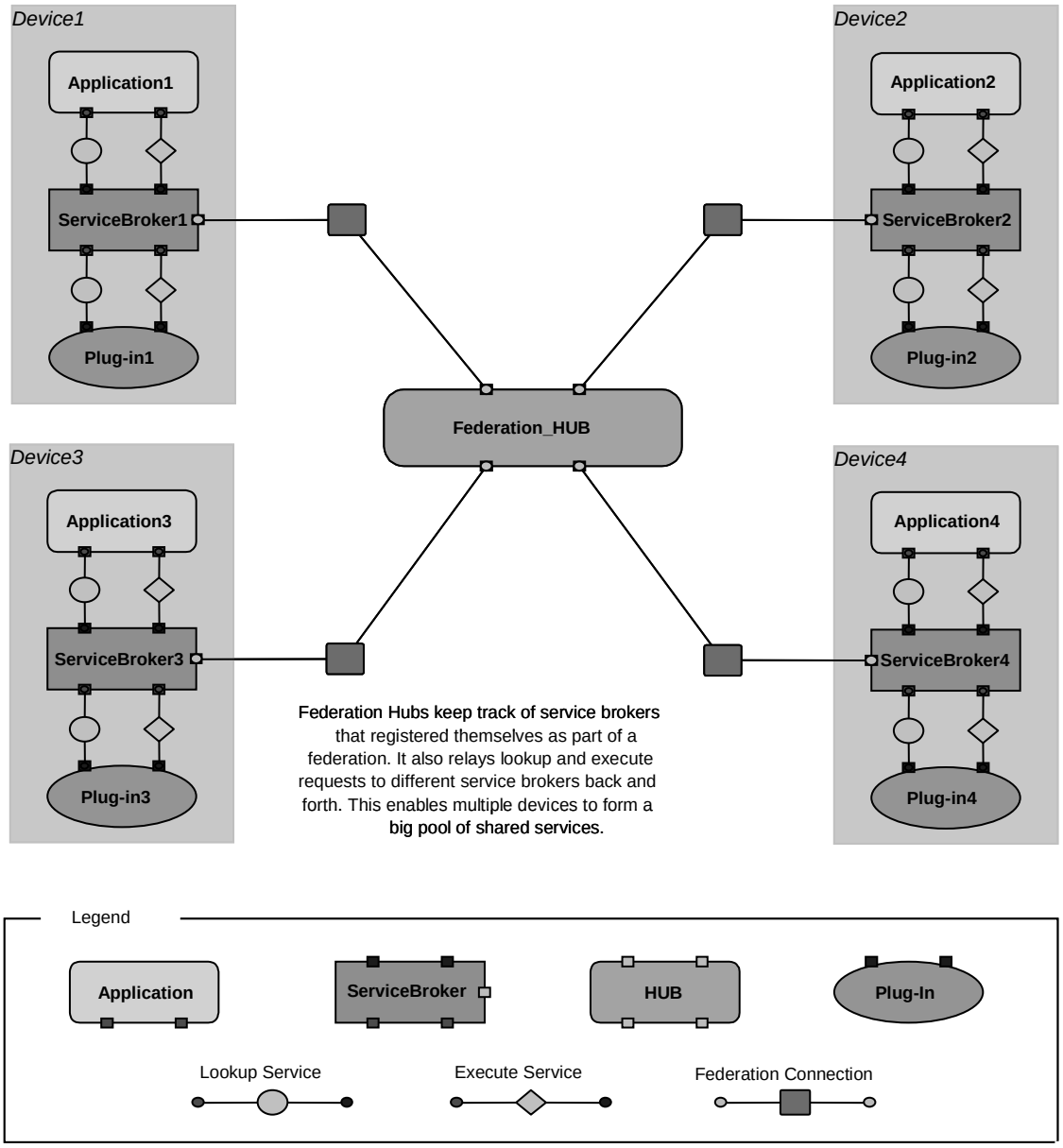


Figure 11: D-PAMD Architecture

Note that there might be multiple federation hubs, each maintaining a single federation. Service brokers can decide which federation to join by selecting from among different hubs. Though we will not discuss the following issues in detail here, they must be addressed to enable this architectural extension:

- design of the new service broker hub
- modification of the existing service brokers
- protocols between service brokers and a hub
- security issues in a federation such as access control and user authentication
- peer-to-peer connection without a federation hub
- modification to the format of plug-in identities (including a device ID)

To allow easy the modification of existing service brokers, an additional federation interface component needs to be added inside the service brokers. This new component works as a bridge between the Federation Hub and internal components inside the service brokers—Discovery, MatchMaker, and Dispatcher. With this approach, existing internal components can be reused as much as possible, and peer-to-peer service sharing can be achieved by replacing federation interface components with peer-to-peer interface components.

There might be a problem with identical plug-ins existing in different mobile devices. For this reason, plug-in identity information should include a unique device identifier.

8 Conclusion

"Jim, Can you send me the financial report from your Palm?"

"Get it from my Palm, Bill. You know the access code."

"Hmm. Yeah, sure Thanks Jim! I wanted to review it and include it in a press release today." (Thanks to PAMD, I can do this easily using my Palm even though I'm traveling.)

This would be the future of PAMD.

PAMD provides interoperability between applications and plug-ins without sacrificing the performance of the mobile devices. Because existing applications can be made PAMD compliant with little modification, the development time and costs of adding functionality to them can be reduced dramatically. As PAMD bears the burden of communicating with plug-ins, application and plug-in developers can develop their own products independently and easily use each other's products. This will certainly create a competitive market to create better plug-ins, thus benefiting the customer.

PAMD satisfies all the architectural requirements we described in Section 2. Though PAMD is developed and tested on mobile devices, especially Palm, it could be adapted to any devices on any platforms that need interoperability. PAMD will handle any failures during the interaction between applications and plug-ins, thus providing reliability. PAMD imposes minimal performance overhead with its efficient lookup and execution process.

In this technical note, we described PAMD, its interfaces, how applications and plug-ins interact with them, and the advantages of using PAMD. Several scenarios were illustrated to explain the architecture and how it can be implemented. We also suggested some extensions to enhance the current architecture. We hope that this work is useful to mobile device application developers, plug-in developers, and manufacturers. We predict that the day when all mobile devices are equipped with PAMD is not far away.

Appendix A Acme Textual Description

```
Family PAMD_Outside = {

    Port Type provide;
    Port Type use;

    Component Type Application = {
        Port lookup  : use;
        Port execute : use;
    };

    Component Type PAMD-Layer = {
        Port identity : use;
        Port service  : use;
        Port lookup   : provide;
        Port execute  : provide;
    };

    Component Type PlugIn = {
        Port identity : provide;
        Port service  : provide;
    };

    Role Type caller;
    Role Type callee;

    Connector Type CallReturn = {
        Role caller : caller;
        Role callee : callee;
    };
};

Family PAMD_Inside = {

    Port Type provide;
    Port Type use;
    Port Type read;
```

```
    Port Type write;

    Component Type Interface = {
        Port provide : provide;
        Port use     : use;
    };

    Component Type Object = {
        Port provide : provide;
        Port use     : use;
    };

    Component Type Repository = {
        Port read  : read;
        Port write : write;
    };

    Role Type source;
    Role Type sink;
    Role Type caller;
    Role Type callee;

    Connector Type Access = {
        Role source : source;
        Role sink   : sink;
    };

    Connector Type CallReturn = {
        Role caller : caller;
        Role callee : callee;
    };
};

System toplevel : PAMD_Outside = {

    Component PAMD-Layer : PAMD-Layer = {

        Port identity : use;
        Port service  : use;
        Port lookup   : provide;
        Port execute  : provide;

        Representation {
```

```
            System inside : PAMD_Inside = {

                Component LookupInterface : Interface = {
                    Port provide : provide;
                    Port use     : use;
                    Port use2    : use;
                };

                Component Discovery : Object = {
                    Port provide : provide;
                    Port use     : use;
                    Port write   : write;
                };

                Component MatchMaker : Object = {
                    Port provide : provide;
                    Port use     : use;
                    Port read    : read;
                };

                Component Dispatcher : Object = {
                    Port provide : provide;
                    Port use     : use;
                    Port read    : read;
                };


                Component ExecInterface : Interface = {
                    Port provide : provide;
                    Port use     : use;
                };


                Connector CallReturn1 : CallReturn;

                    LookupInterface.use to
CallReturn2.caller;
                    MatchMaker.provide  to
CallReturn2.callee;

                    ExecInterface.use  to
CallReturn3.caller;
```

```
                    Dispatcher.provide to
CallReturn3.callee;

                };
            };

            Bindings {
                execute to ExecInterface.provide;
                service to Dispatcher.use;
                lookup to LookupInterface.provide;
                identity to MatchMaker.use;
            };
        };
    };

    Component Application_1 : Application;
    Component Application_2 : Application;
    Component Application_N : Application;

    Component PlugIn_1 : PlugIn;
    Component PlugIn_2 : PlugIn;
    Component PlugIn_N : PlugIn;

    Connector CallReturn1  : CallReturn;
    Connector CallReturn2  : CallReturn;
    Connector CallReturn3  : CallReturn;
    Connector CallReturn4  : CallReturn;
    Connector CallReturn5  : CallReturn;
    Connector CallReturn6  : CallReturn;
    Connector CallReturn7  : CallReturn;
    Connector CallReturn8  : CallReturn;
    Connector CallReturn9  : CallReturn;
    Connector CallReturn10 : CallReturn;
    Connector CallReturn11 : CallReturn;
    Connector CallReturn12 : CallReturn;

    Attachments {
        Application_1.lookup  to CallReturn1.caller;
        Application_1.execute to CallReturn2.caller;
        Application_2.lookup  to CallReturn3.caller;
        Application_2.execute to CallReturn4.caller;
        Application_N.lookup  to CallReturn5.caller;
        Application_N.execute to CallReturn6.caller;
```

```
        PlugIn_1.identity to CallReturn7.callee;
        PlugIn_1.service  to CallReturn8.callee;
        PlugIn_2.identity to CallReturn9.callee;
        PlugIn_2.service  to CallReturn10.callee;
        PlugIn_N.identity to CallReturn11.callee;
        PlugIn_N.service  to CallReturn12.callee;

        PAMD-Layer.lookup to CallReturn1.callee;
        PAMD-Layer.lookup to CallReturn3.callee;
        PAMD-Layer.lookup to CallReturn5.callee;

        PAMD-Layer.execute to CallReturn2.callee;
        PAMD-Layer.execute to CallReturn4.callee;
        PAMD-Layer.execute to CallReturn6.callee;

        PAMD-Layer.identity to CallReturn7.caller;
        PAMD-Layer.identity to CallReturn9.caller;

        PAMD-Layer.identity to CallReturn11.caller;

        PAMD-Layer.service to CallReturn8.caller;
        PAMD-Layer.service to CallReturn10.caller;
        PAMD-Layer.service to CallReturn12.caller;
    };
};
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	August 2002	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Plug-In Architecture for Mobile Devices	F19628-00-C-0003

6. AUTHOR(S)

Madhu Keshavamurthy, Jung Soo Kim, Mona Li, Vichaya Sagetong

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2002-TN-023

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

This technical note describes plug-in architecture for mobile devices (PAMD)—an architectural specification that extends the function of applications in mobile devices. Users gain major benefits when the functionality of applications that run on these devices can be extended through the addition of new services that don't require changes to the application itself. PAMD provides interoperability between applications and plug-ins without sacrificing the performance of the mobile devices on which they run. Because existing applications can be made PAMD compliant with little modification, the development time and costs of adding functionality to them can be reduced dramatically. As PAMD bears the burden of communicating with plug-ins, application and plug-in developers can develop their own products independently and easily use each other's products.

This technical note also describes PAMD's interfaces, how applications and plug-ins interact with them, and the advantages of using PAMD. Also included are several scenarios that explain the architecture and how it can be implemented, and suggestions for extensions that enhance it.

14. SUBJECT TERMS	15. NUMBER OF PAGES
PAMD, plug-in architecture for mobile devices, plug-in	44

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL