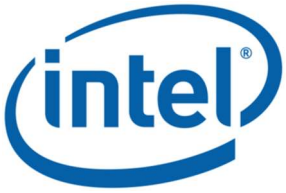

Intel® QuickAssist Technology Accelerator Abstraction Layer Software Programmer's Guide

---

Revision 3.00.27

2012

Authors: Joe Grecco

Aaron Grier

Neal Oliver

Contributors: Bhushan Chitlur

Henry Mitchel

Tim Whisonant

Notices

Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license. Specifically:

This document contains information on products in the design phase of development.

Document Change Log

Send proposed changes and comments to Joseph Grecco IAG/DSCG/IDP/ATP
joe.grecco@intel.com

Version	Date	Changes	Author
1.00	3/31/2009	Release for 1.00	H. Mitchel, J. Grecco
2.00.00	10/15/2010	Add AALASM Documentation Updated for QPI Interface changes 1.10 → 2.00	A. Chen H. Mitchel
3.00.00	10/08/2012	3.00 re-write	J. Grecco, A. Grier

Table of Contents

Table of Figures

1 About this Document

Intel QuickAssist Accelerator Abstraction Layer (AAL) Technology provides a rich set of runtime and software development tools that facilitate the deployment of systems consisting of a collection of non-uniform, asymmetric compute resources.

The purpose of this document is to provide the reader with a detailed description of the AAL concepts, architecture, and design elements. Its intended audience is system engineers, platform architects, and software developers.

The document is organized into a series of chapters that progressively dive deeper into the technical details of AAL. The first chapters cover the rationale behind AAL and the concepts upon which AAL is based. They are intended to provide system engineers and platform architects a sufficient understanding of AAL to determine whether this technology is applicable to their needs. The remainder of the document serves as a primer to the software developer preparing to use the AAL SDK and runtimes. This document is not intended to be a software reference. Software developers are directed to the AAL SDK reference documents for a detailed description of AAL's development tools and libraries.

2 Introduction

A case is made for a common accelerator abstraction library, and Accelerator Abstraction Library (AAL) is introduced as a solution. The general architecture of AAL is introduced, interactions between architectural elements are described, and deployment examples are provided.

2.1 The case for an accelerator abstraction library

Servers, clusters, and other multi- or many- resource systems differ from one another in many ways, but one of the most important is the symmetry and uniformity of the resources that make them up. These attributes determine fundamentally how the resources are managed by system owners, and used by developers.

At one end of the spectrum are symmetric multi-processing (SMP) servers, by their very name very symmetric and uniform. Because of their symmetry and uniformity, tasks running on such a system can be scheduled efficiently by an operating system, and developers can write software using a very abstract model, such as threads, without needing to know any details of the underlying hardware (in fact, they may choose to ignore even this model and write single-threaded code, with no functional impact on their software, only potential performance impact).

Systems can break symmetry and uniformity in a variety of ways, e.g.: by mixing processors with different instruction sets; creating memory banks or caches with non-uniform access properties; mixing CPUs, GPUs, reconfigurable devices, and fixed function devices; and by interconnecting resources in asymmetric graphs.

Breaking symmetry and uniformity is approached only with great fear and trepidation by server architects, because operating systems lose the architectural models that they use to schedule software on the system. Upgrading operating systems and software development tool flows to understand the new models is still basically a research problem, so the architectural models must be presented to the system manager and the software developer.

Despite the cost, symmetry and uniformity is still broken when customer usage models force it to happen for reasons of performance. The most common use-case is that a special-purpose device must be used to provide a function that cannot be replicated by a CPU, but there are many other cases. These usage models are the area in which AAL operates.

In the AAL conceptual model, a server consists of a collection of non-uniform, asymmetric resources, each potentially with their own properties. An application program uses the server by requesting a collection of resources that it requires, receiving the exclusive use of those

resources for a time, executing a program, and deallocating the resources. The resources are expensive, in the sense that it is infeasible to provision the system with more instances than will ever be used by a system, and must be allocated in an exclusive manner, requiring an application and a resource manager to interact.

This model is essentially the simplest possible model for an asymmetric, non-uniform system architecture. In the Intel world, it was first developed and deployed in the mid-1990's in the domain of computer telephony, at the time an environment where the computing demands of media processing (speech recognition, text-to-speech, call control) required discrete cards for media processing offload; and currently, the same model is being applied for servers with attached GPUs and FPGAs. This model was independently developed for OpenCL, the architectural model of which consists of processing elements (PEs) allocated into processing groups and devices.

This model, and middleware such as AAL, fills a need when a customer usage model cannot be accommodated by existing operating system and development tool technology. The use of the model in computer telephony was eventually replaced by technology such as DirectX, in which nodes of a media processing graph were scheduled by the Windows kernel; but this happened only years after computer telephony products based on this model were deployed, and only after companies such as Intel upgraded processor technology to meet the processing requirements. Today, the discrete devices deployed with servers are GPUs, reconfigurable devices such as FPGAs, and fixed-function devices such as ASICs, and the usage models supported are for high performance computing, computer graphics and vision, and network communications; but the AAL conceptual model applies.

Conclusion

The differentiating value of AAL lies in its minimal model of an asymmetric, nonuniform server architecture for both the system manager and the developer. This model is essential whenever a customer usage model demands computing or communication resources that cannot be accommodated by a CPU scheduled by a current-generation operating system. It allows a customer usage model to be accommodated while research to improve hardware, operating system, and software development tools is carried out.

3 Introduction to the Accelerator Abstraction Layer Runtime Framework (AAL)

Intel QuickAssist Accelerator Abstraction Layer (AAL) is a runtime framework that implements a model of Services deployed across an asymmetric collection of non-uniform compute resources. A **Service** is an encapsulation of functionality that consumes **compute resources.** Compute resources include, but are not limited to, General Purpose Processors (CPUs), purpose built fixed function coprocessors, and programmable accelerators. Services provide useful work to **client applications** through a Service-specific **application programming interface** (API).

AAL's facilities allow Services to be implemented in a variety of ways while abstracting the Service implementation from the client application. Service implementations can be created that utilize the best possible compute resources available on the platform. Provided the Service API remains the same, applications can transparently use Services having vastly different implementations. This abstract service model allows applications and Services to be deployed across a wide variety of platform configurations.

3.1 Service-Oriented Architecture

AAL is based on an **Object-Oriented** and **Service-Oriented Architecture** (SOA). Examples of SOA based designs are in widespread use. For example, SOA forms the foundation of many of the web-based facilities in popular use today.

3.1.1 Attributes of a Service-Oriented Architecture

For the purpose of this discussion, a Service-Oriented Architecture may be characterized as having the following properties:

- **Services** – One or more executable Objects[1], consuming compute resources and implementing a rigorously defined API.
- **Registrar** – A facility used to register services and their APIs. Also used to locate and acquire a Service's Interface.
- **Clients** – One or more executable entities that utilize Services.
- **Transport** – A mechanism by which entities communicate. This can be anything from direct function calls to traditional networking technology (e.g., TCP/IP)

[1] We use the term "Object" here loosely to mean an executable entity that presents an API.

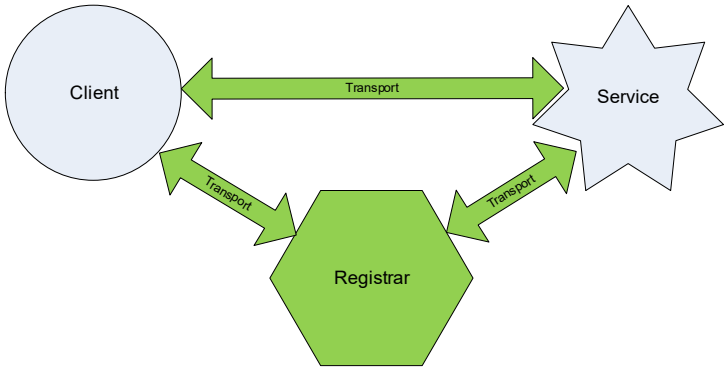

Figure 1 – Basic SOA Model

In the basic SOA flow, Services are registered with the Registrar, indicating the API they implement. Clients make use of Services by acquiring the Service's API through the Registrar. This model is similar in concept to an application utilizing a shared- or dynamically-linked library of functions. Function interfaces are implemented in Objects and are linked at runtime.

However, the underlying technology used by AAL is quite different. In traditional dynamic linking, the application has embedded symbolic information about the library it is loading. This information is encoded into the application at build time. In general there is a static cardinality of 1 between the application and the library that provides the API implementation. That is, in general there can be only 1 implementation of a particular Service API linked with the application at a given time. In contrast, client applications in AAL are not "linked" to the API implementation in the traditional way.

When an application requires use of a Service it uses AAL facilities utilizing the Registrar. These facilities locate one of potentially multiple implementations of the desired Service. The application is returned the Service interface at runtime using a **late binding** technique described in more detail later.

The net result is that applications are more loosely coupled to the Service implementation than through traditional linking. There are no longer build time dependencies between the client application and the Service implementation. Link time symbol resolution is no longer performed, eliminating many problems associated with incompatibilities with library implementation changes (versions) and namespace collisions. Applications can even utilize multiple distinct implementations of the same Service API simultaneously.

3.2 Platform Services

As a Service Oriented framework, AAL is easily extended and customized by adding and replacing Services. In general, an AAL-based platform can be thought of as a collection of Services registered with a Registrar and connected via a logical network.

AAL defines a core set of facilities that implement the Service Oriented infrastructure (e.g., Registrar and Transport). AAL defines a set of additional facilities that extends the basic Service-Oriented capabilities. These additional facilities make up the **Platform Services** present on the system. In an AAL based implementation, there are mandatory Platform Services and optional Platform Services. Most Platform Services, even some mandatory ones, can be replaced with customized implementations[2]. Platform Services differ from any other Service only in that a mandatory interface (API) for the Service is specified by AAL.

The set of additional Platform Services defined by AAL are:

- **Service Factory** – Used by the client application to request a Service.
- **Event Delivery** – Specialized transport service used for intra-process message delivery and functor[3] scheduling.
- **Resource Manager** – Used to manage certain compute resources used to implement Services.
- **Policy Manager** – Service used to decide how and when compute resources may be used to construct a Service.
- **Reconfiguration** – Service used to change the configuration of compute resources during the instantiation of a Service.

Each of these services will be discussed in more detail in future sections.

3.3 Object Communications and Events

As described above, the basic flow of a SOA is that a client application requests the interface to a Service and then invokes **methods** (i.e., functions) on that interface to perform work. AAL defines a communications model that applications and Services conform to. The first aspect of

[2] In theory any facility can be replaced in AAL. For practical reasons some facilities have not been designed for easy replacement.

[3] A "functor" is a type of object that is capable of executing a function call.

the model to describe is the **call model** (i.e. the way Service API functions are called). AAL defines an **asynchronous** call model for applications invoking methods on a Service. In a **synchronous** call model, when an application invokes a method on a Service, the application's thread of execution blocks inside the method call until the requested operation completes. In contrast to a synchronous model, when an application invokes a method asynchronously, the thread of control returns to the application immediately while the requested operation executes in parallel. Section 3.4 discusses the call model in more detail.

Where the call model describes how the application communicates with the Service, the other aspect of the communications model has to do with the ability for Services to communicate back to the application. The following section goes into detail of how the application and Service inter-operate when the application requests work to be done.

In general, the Service model is often referred to as **client/server** in that operations are initiated by the client via a **request** and the Service (i.e., server) returns **responses** to that request. However, there may be scenarios where the Service needs to contact the client application of its own accord (i.e., **unsolicited**), for example to report an exception or system error.

AAL defines a mechanism for delivering **Events** to the application. Both solicited responses and unsolicited notifications may be reported via Events. AAL Events are formalized objects that implement an AAL-defined interface. Services may extend the basic AAL Event by adding its own interface to the Event object. Events are delivered to the application using the AAL **Event Delivery Service (EDS)**. AAL requires that client applications supply a default **Event Handler**, a function used to handle events, when acquiring a Service.

While the default mechanism for communications between Service and client is through Event Handlers, we'll see later that a "peer-to-peer" communications model can also be constructed that blurs the distinction between client and Service. This will be discussed in more detail in future sections.

3.4 Transactions and the Asynchronous Call Model

Service methods that initiate work on behalf of a client always result in a notification back to the client when the work is completed. In AAL we refer to the asynchronous operation being performed by the Service as a **Transaction**. Transactions are stateful; they begin life with the invocation of a Service method and they end with a completion notification. While a Transaction exists it is considered to be **active**. Multiple Transactions may be active in a Service simultaneously. Client Applications may have multiple Transactions active on the same or

multiple Services simultaneously. This allows the application to perform multiple operations simultaneously, or stated differently, more work can be performed at the same time.

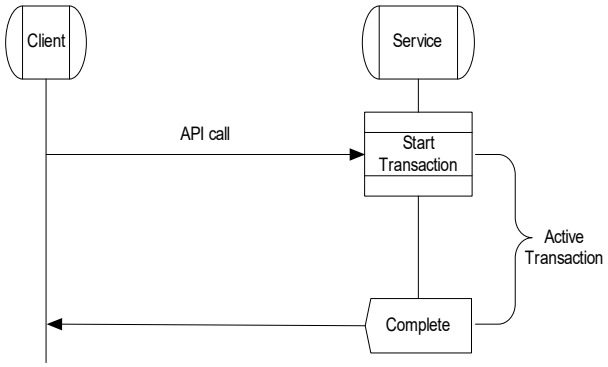

Figure 2 – Transactions

Because the Service's operation runs asynchronously, AAL must define a mechanism that allows the application to know when the operation has completed. While active, Transactions may go through an arbitrary number of state changes. By definition a Transaction goes through at least 1 state change and that is from **Active** to **Completed**. To properly process a Transaction, the Service may be required to notify the application of intermediate state changes that occur while the Transaction is active. In general, AAL defines 2 models for handling the asynchronous processing of Transactions. As mentioned above, Services may report Transaction state transitions by generating Events (See Section 3.3). This model of Transaction handling is referred to as the **Event Model**.

Another model for processing Transactions involves the client application exchanging an interface of its own with the Service. Rather than an AAL Event object being delivered through the EDS, the Service invokes methods directly on the client application to notify the client that a Transaction state change has occurred. This model is referred to as the **callback** or the **consultation** Model. Directly calling into an object's interface in an asynchronous model can be problematic. Consider a recursive callback scenario where object A calls Object B which calls back object A which, in-turn, calls back object B, all in the same thread. AAL provides a mechanism through use of special "functor" events, which allow Services to safely schedule a callback using the system's Event Delivery Service.

3.5 Services, Interfaces and Composition

We've touched upon the fact that AAL Services define Interfaces that are registered with the Registrar facility and then later used by client applications to perform work. This section examines the design pattern used by AAL for implementing Service Interfaces.

AAL employs a variation of the **Composite** and **Delegation** design patterns to implement Services. Interfaces are defined by the Service designer with no implementation details exposed. An example in C++ uses a **pure virtual class** with no **private** or **protected** methods and no member variables.

```
class IMyInterface{
   virtual doThis(void)=0;
   virtual doThat(void)=0;
   virtual ~IMyInterface(){}
};
```

Figure 3 – Sample Interface

Component objects contain the implementation of the interfaces. The interface or a method of an interface may be implemented directly by the Component object or may be implemented by "child" objects that the Component aggregates. The Component may also perform partial functionality of a method and **delegate** the remainder of the functionality to another object (or Service).

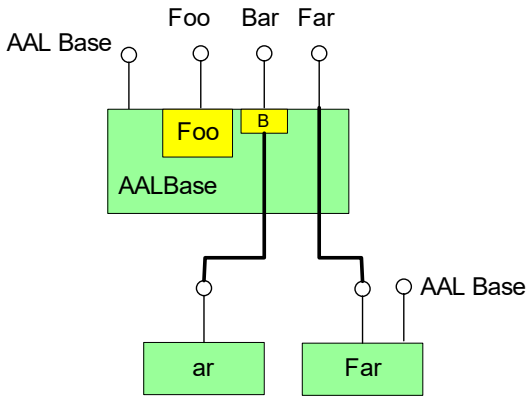

Figure 4 – Interfaces, Objects and Aggregation

Figure 4 shows a Component based Service that exposes multiple interfaces. In this example the Service implements the Foo interface internally, perhaps as a built-in object. Interface Bar is partially implemented in the Service parent (block labeled B) but the remainder is implemented in the object labeled "**ar**", finally the Interface Far is entirely implemented by the delegate Service labeled **"Far"**. The AAL **Base class** interface will be discussed in detail later in this

document, but this example is intended to illustrate that "Far" may be a registered Service in its own right.

Client applications are insulated from this level of the Service's implementation detail. The client accesses the Service through the exposed interfaces which do not expose implementation. AAL provides facilities that permit the application to access the interfaces of the Service uniformly. For example, the code an application would use to access the Far interface in the above Composite Service would be exactly the same as if it were accessing the Far interface directly on the Service labeled "Far". How these mechanisms work in detail will be described in future sections.

This design pattern enables many powerful capabilities. As was mentioned in Section 3.1.1, AAL uses a method of "**late binding"** to interfaces. That means that the client gets the concrete implementation of an interface at the time it requests the interface from the Service. This late binding allows the Service to potentially select the implementation it returns at the last minute, possibly using a heuristic to decide which implementation to return or even load from another Service.

3.6 Factories, Resources, and Resource Management

AAL is built upon a Service-Oriented Architecture. Section 3.1 describes the characteristics that make up a basic SOA platform. In Section 3.2 we explained that AAL extends the functionality of a basic SOA framework by defining **Platform Services**. AAL defines two powerful features; the ability to abstract the construction (i.e., instantiation) of Services, and the ability to manage the compute resources used by the implementation of the Service.

By abstracting the instantiation of the Service, AAL enables Services to be created dynamically, only using compute resources when they are required. In the case where multiple implementations of the same Service are available, AAL can return the instance that best meets the needs of the platform. AAL's **Resource Management** Platform Services allows the system to intelligently control the provisioning (i.e., configuring) and allocating of compute resources to Services. This can be particularly powerful when some of those compute resources are shared and precious, such as a programmable accelerator or FPGA.

AAL implements a **Factory** design pattern to abstract the instantiation of Services from the application. The **AAL Service Factory** provides the interface typically used by client applications to request a Service. The Service Factory interacts with the Registrar, usually through the Resource Manager Service, to determine how a Service is to be instantiated.

For example, consider the platform described in Figure 5. The box labeled **Service Host** represents the compute resources used to implement the Service. $Host_{A,}$ in red, represents

some precious accelerator resource such as an FPGA. $Host_C$ represents the host CPU. $Service^x_{Foo}$ represents some Service exposing an interface "Foo".

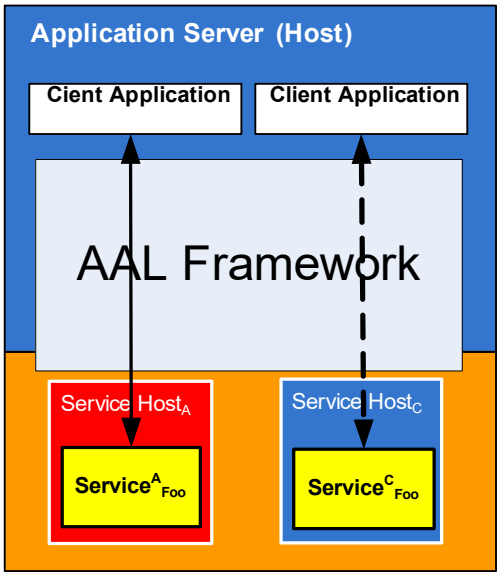

Figure 5 – Example of Multiple Service Instances

For this example, there exist 2 implementations of $Service_{Foo}$. One is implemented on the accelerator and one is implemented on the CPU. Because both Services expose interface Foo, performance notwithstanding, the two are identical to the client. For the purpose of this example there are 2 applications that wish to use $Service_{Foo}$. The first client application is granted the implementation on the accelerator. The second application, rather than being denied the Service, is granted an instance constructed using the CPU.

In the above, simple example, the client applications consulted the Service Factory requesting a $Service_{Foo}$. The Service Factory consulted the Resource Manager Service which determined the available implementations and which compute resources could be used to provide $Service_{Foo}$. The Resource Manager Service returns the information required to allow the Service Factory to instantiate the requested $Service_{Foo}$.

This is a simplified example. On a sophisticated platform the Resource Manager Service may have consulted a Registrar to determine the possible ways the Service could be constructed given the platform resources and installed software. The Resource Manager could then take that information combined with the current resource allocations and consult a **Policy Manager** Service to determine which implementation should be instantiated. The instantiation of the Service is orchestrated by the Service Factory using the information returned by the Resource Manager. The actual instantiation, which could involve any number of operations, is delegated to a specialized "Concrete" Factory designed for the task.

Not all platforms require this level of sophistication. Because AAL is largely made up of a suite of loosely coupled Services, systems can be constructed that have only the functionality required to meet the system requirements.

3.7 Summary

AAL is a runtime framework designed for building performance optimized platforms by implementing a model of Services instantiated across a collection of non-uniform compute resources.

Based on a Service Oriented foundation, AAL is fundamentally flexible, scalable, and adaptable to a wide variety of usage models. Expanding on the Service Oriented core facilities, AAL defines additional Platform Services that provide a number of advanced features such as Resource and Configuration Management Services. Because AAL is simply a collection of loosely-coupled Services, Platform Services can be replaced, extended, added or in many cases omitted as dictated by the system requirements.

Service Interfaces are provided using a "**late binding**" technique rather than "**link-time binding**", enabling clients to transparently use different implementations of the same Service interface, even simultaneously. Combined with the Factory abstraction of Services instantiation, the system can dynamically allocate and, if need be, re-provision compute resources on the fly to create the most optimal mapping of resource to Service.

This provides the application with a consistent abstraction to accelerator functionality. The functionality may be implemented in hardware, pure software, or a combination of hardware and software. The application is insulated from the implementation details of the functionality it is using.

Each hardware resource may provide an individual exclusive service, like a fixed-function accelerator, or may be reconfigurable through an associated reconfiguration service.

4 Theory of Operation

In Section 3 we described AAL at an architectural level. The principles of a Service-Oriented Architecture were described, as well as how AAL is fundamentally a Service-Oriented runtime framework. We introduced the concept of Services as one or more executable objects that consume compute resources and perform work on behalf of a client application. Platform Services, as a means of defining additional functionality, were discussed in section 3.2. Service Interfaces and the asynchronous call model were described in sections 3.3 - 3.5. Finally an example of how a simple flow could utilize all of these elements was described in section 3.6.

The purpose of this chapter is to continue to the next level of detail to examine the system design. Because of the fundamentally dynamic nature of a framework like AAL, and the ever-changing details of AAL implementations, a detailed discussion of how a particular AAL implementation is designed is beyond the scope of this document. The curious reader should refer to reference documents and even the source code as the ultimate authority. However, the theory and design of the AAL system will be discussed in sufficient detail as to allow the reader to effectively use and even examine the source code of an AAL platform.

4.1 Service Domains

In Section 3.1.1 the basic model of a SOA platform was described. One or more Services register with a Registrar. Clients discover and acquire interfaces to those Services through the Registrar. The whole system is connected via a transport technology.

AAL introduces the concept of **Service Domains** to the basic SOA model. A Service Domain can be thought of as the collection of Services associated with a single Registrar. A Service Domain contains one and only one Registrar. An example of a Service Domain in AAL is the collection of all Services implemented as **User Mode** executables on a single host platform. Another example of a Service Domain is the **Kernel Mode Service Domain**. The Kernel Mode Service Domain is made up of a collection of kernel mode (e.g., ring 0) Services and clients. The User and Kernel Mode Service domains will be discussed in more detail in the next sections.

Clients generally interact with a single Registrar and, as such, can be considered to be **local** to that Service Domain.

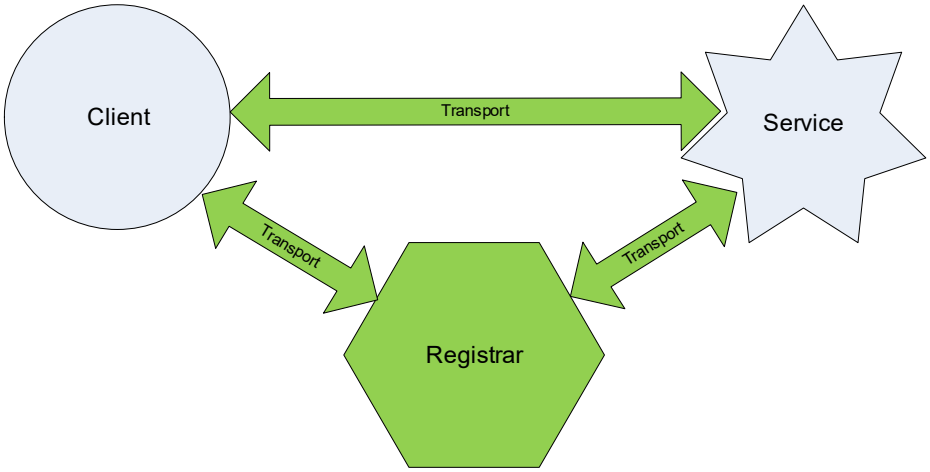

Figure 6 – SOA Model (Service Domain)

AAL allows Clients to access Services that reside in a foreign domain through the use of **Service Domain Bridges**. Bridges generally implement protocols enabling Registrars of different domains to cooperate. In addition, bridges may act as interface adapters, providing marshaling and un-marshaling services for Service APIs.

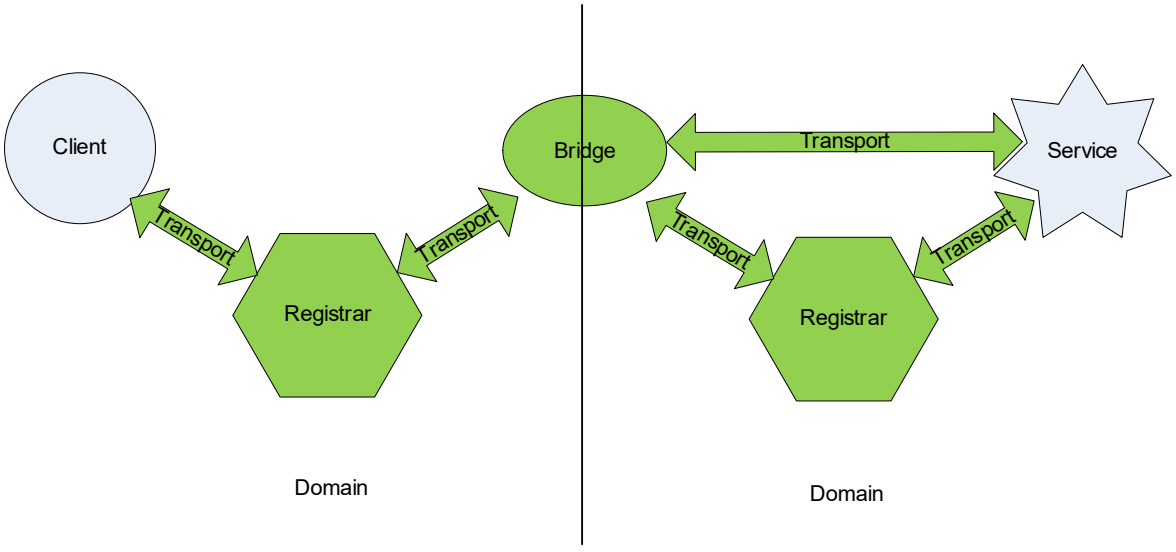

Figure 7 – Bridging Service Domains

Service Domain Bridges are not generalized in AAL. Implementations of bridges are optimized on a case-by-case basis. There are several examples of Service Domain Bridges in AAL. One that

will be discussed in more detail later is the User/Kernel Domain Bridge implemented by the **Application Interface Adaptor** (AIA) and **User Mode Interface Driver** (UIDRV). This bridge allows user mode client applications to access Services that utilize accelerator hardware through kernel mode i.e., ring 0) device drivers.

Clients are insulated from the Domain(s) a Service appears on. Regardless of where a Service implementation resides, its interface is registered with the local Registrars of the Domain(s) on which it wishes to be accessed. Clients acquire these Services through the AAL Factory Platform Service in the usual way. The Client accesses remote Services via a Service Proxy which implements the local part of the Service Interface.

4.2 User Mode Service Domain

The **User Mode Service Domain** (UMSD) is the domain in which most developers operate. As the name implies, the User Mode Service Domain executes in user mode (i.e., ring 3) of the host processor/OS.

Clients operating in the UMSD generally run in the context of an **OS Host Process**. In section 3.2 we described how AAL extends the basic SOA model with Platform Services. These Platform Services include the Service Factory and Event Delivery service. Clients access AAL's core User Mode Platform Services, also known as the **AAL Abstraction Services** (AAS) by linking to the AAL runtime library. A single OS process may contain multiple clients; however all clients within a process share a single copy of the AAL User Mode Platform Services.

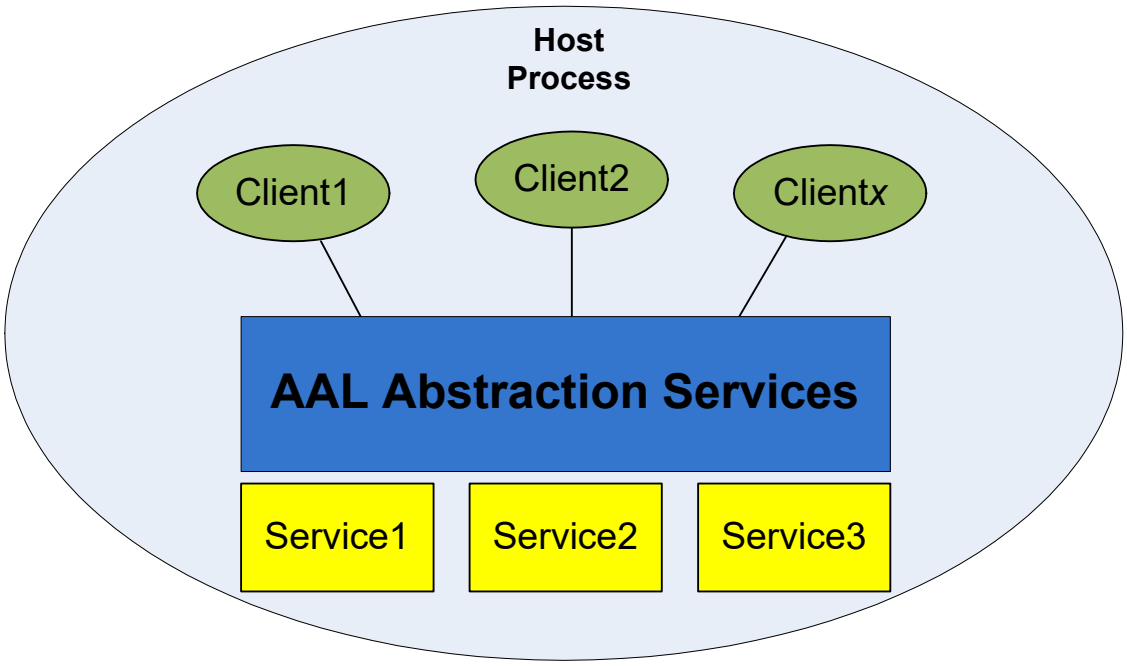

Figure 8 – Clients in a Process

Services, with few exceptions[4], are implemented as dynamically-linked libraries. When a client requests a Service using the AAS Service Factory, the Factory ultimately loads the Service implementation into the client's process address space.

Figure 8 illustrates the physical composition of an AAL application. The large oval represents an OS process space. The smaller ovals labeled "Client" represent codes that implement a client flow. Clients may share the same thread or may execute in separate threads. The rectangle labeled AAL Abstraction Services represents an instance of the library implementing the AAS. Finally the smaller rectangles labeled Service*X* represent individual Services that have been loaded into the Host Process address space.

[4] Some of the core AAS Services are currently built into the AAL runtime library.

AAL supports a wide variety of configurations. The following figures show some example deployments. Figure 9 represents a configuration where each client application is contained in a separate process and the entire platform resides on a single host node.

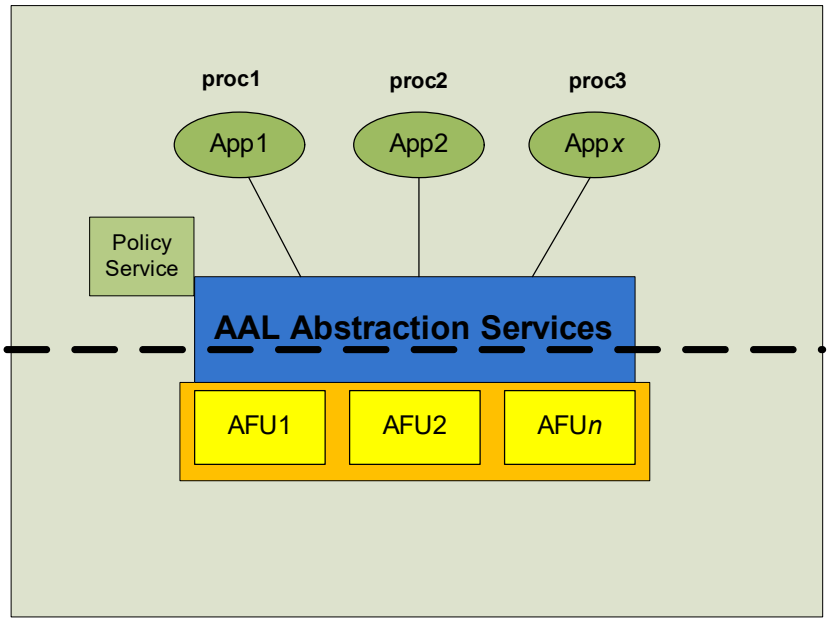

Figure 9 – AAL Single Server

Figure 10 depicts a multi-node system with a centralized resource management service.

Figure 10 – AAL Multi-node with Central Resource Manager

Figure 11 shows Services connected to remote nodes via a traditional network transport.

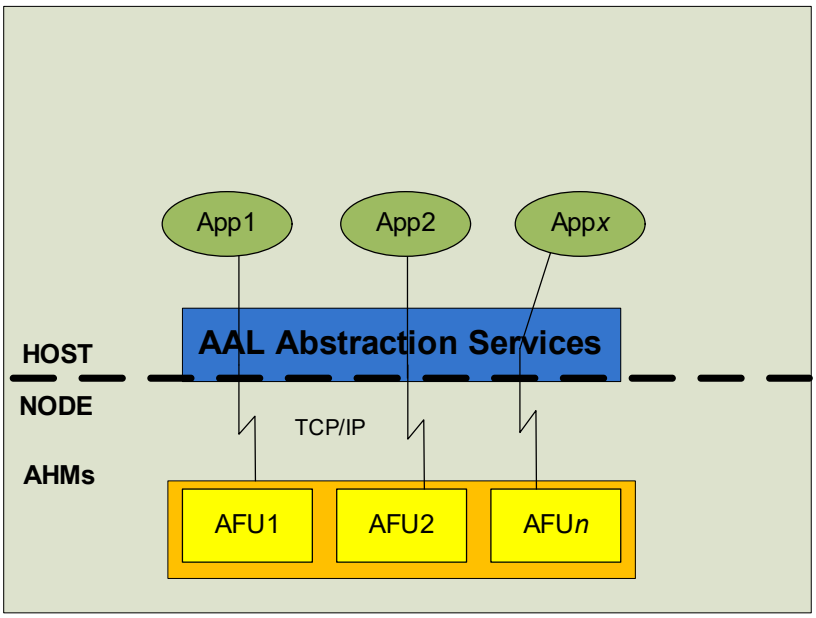

Figure 11 – AAL with Remoter Resource Nodes

Services in the User Mode Service Domain may utilize hardware accelerators directly using proprietary methods (i.e., outside of the AAL framework), such as accessing the hardware through preexisting device drivers and libraries. Services may also access hardware accelerators via AAL Service proxies through the Kernel Mode Service Domain. In the latter case the hardware accelerator is more tightly integrated into the AAL framework and may take advantage of additional AAL facilities for resource management and dynamic re-provisioning.

AAL does not require hardware accelerators to be tightly integrated into the framework to be available for use on the platform.

4.2.1 Platform Services

This section describes in more detail the standard Accelerator Abstraction Services.

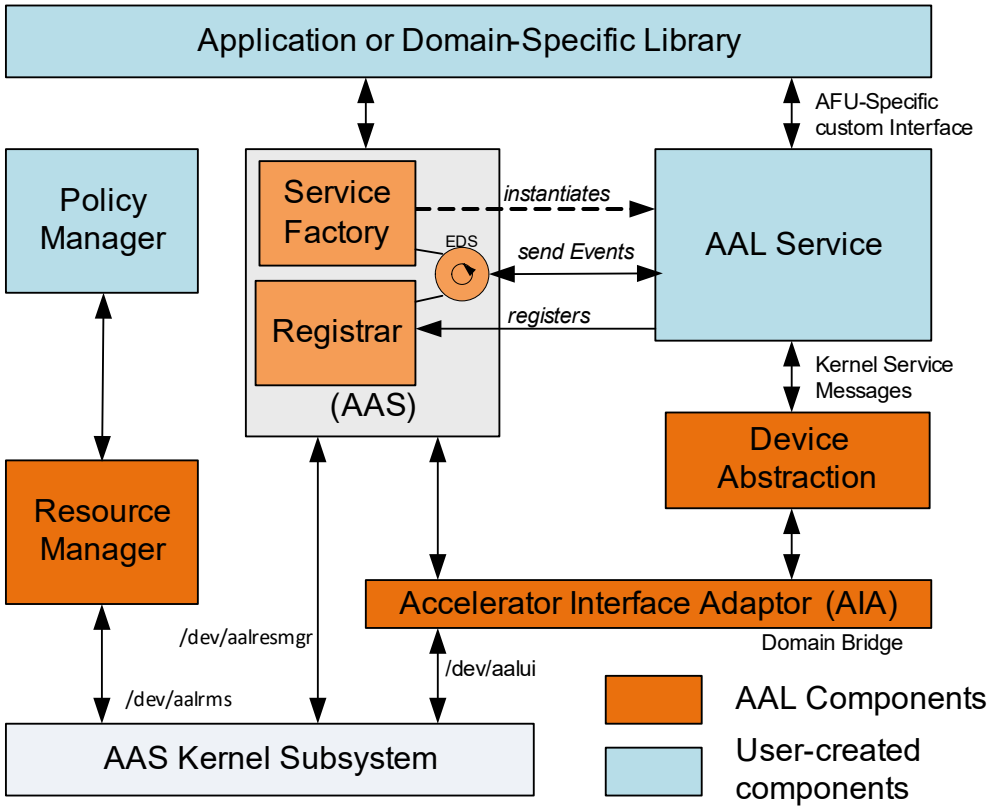

Figure 12 – AAL Architecture

In Figure 12 the dark colored blocks depict the Platform Services provided in the AAL User Mode Domain.

4.2.1.1 AAS Core

Before we can get into the details of the AAS Platform Services, we must spend a moment to discuss the AAS Core subsystem. As mentioned in the previous sections, each OS Host Process loads a local copy of the AAS Core services. Before clients can begin to use AAL, the AAS Core subsystem must be initialized by the OS Host Process. This is accomplished by calling the **AAL::SystemInit()** function. The AAS Core subsystem can be thought of as a meta service in that it serves as a container for all services accessed through AAL. As with all Services in AAL, the client provides a default Event Handler (See section 3.3) when initializing the AAS Core. What is returned is an object called the AAS **ServiceContainer**. The ServiceContainer holds references to all of the available AAS Core services. Core services use the AAS default Event Handler when communicating with the application.

When initializing the AAS Core subsystem, the application may provide optional, system-level parameters that affect the global behavior of the system. Refer to the User's Reference for more details.

When the application (i.e., Host OS Process) is finished using AAL it should issue an **AAL::SystemStop()**.

4.2.1.2 Registrar and Resource Manager

The AAS Registrar Service is a simple database that contains all of the configuration information of the system.

A primary responsibility of the Registrar is to maintain configuration information for the Services installed on the platform. This includes:

- Attributes that may be used to identify the Service E.g., name
- Name and location of the executable implementation of the Service's API
- Information used as selection criteria e.g., time of day a Service may be used
- Implementation-specific data used by the Service at runtime

The Registrar is a replaceable and augmentable Service. In its simplest form it is little more than a very basic database. The Registrar Service simply performs simple data operations like adds, deletes, and searches on the database. The Registrar does not "understand" the data it is manipulating. A more sophisticated "augmented" Registrar could interoperate with other Services. For example an implementation of an augmented Registrar interacts with the **Resource Manager Service**. The Resource Manager is a front-end Service that manages compute resources used by other Services. The Resource Manager Service is never used directly by clients and is replaceable. The details of the Resource Manager are specific to the implementation in use. In a standard implementation provided with AAL, the Registrar forwards certain queries, like Service name look-ups, to the Resource Manager Service. The Resource Manager in turn may use the **basic Registrar** to do database look-ups to determine if the Service is creatable, perform resource provisioning and anything else necessary to make the Service available. The Resource Manager then returns the results of the query to the augmented Registrar.

Independent of what implementation(s) are present, AAL publishes a standard API for the Registrar Service. Clients and Services are free to use the Registrar for their own purposes.

Note that applications do not load Service implementations by going directly to the Registrar but rather uses the AAL Service Factory (See Section 4.2.1.3) to instantiate Services.

4.2.1.3 Service Factory

The definition of the basic SOA model we described in section 3.1.1 does not specify how Services are instantiated and compute resources allocated. The AAS Service Factory extends the basic SOA model, implementing a **Factory** Design Pattern**.** The Factory Pattern abstracts the details of how Services are instantiated while presenting a common interface to the client application.

The AAS Service Factory is a mandatory core service responsible for instantiating all other User Domain Services in an AAL platform. The AAS Service Factory is instantiated when the OS Host process initializes the AAS Core subsystem with **AAL::SystemInit()**.

During the creation of a Service, rather than going directly to the **Registrar** to locate a Service, applications use the AAS Service Factory to locate and potentially instantiate new Services. The AAS Service Factory is responsible for accepting Service Creation requests from the application and then interacting with the Registrar and other Platform Services to locate and instantiate the Service that has been requested. The use of the **"Factory Pattern"** to instantiate Services enables the platform to have more control over the allocation and use of compute resources. During the creation of a Service, the Factory can use other Platforms Services, such as the Resource Manager, to re-provision compute resources "on the fly".

Clients request Services by providing a **Service Manifest** to the Factory. The Service Manifest is a record of **Named Value Sets,** an object used in AAL APIs to pass structured data. For details on Named Value Sets (See section 4.4.1). The Service Manifest is opaque to the Factory. It is designed to provide the platform and Service developer a rich and flexible mechanism to allow clients to request and initialize Services. The Manifest contains descriptive information used for locating and initializing the Service. It may be used by several objects involved in the instantiation of the Service including the Service itself. For example, the Manifest includes information used by the Registrar to identify a Service class, such as the Service's name. The Manifest may also contain information used by the Service itself, such as initialization parameters or input arguments. The content of the Service Manifest is dependent on the Service definition as well as how the system Registrar is configured (see section 4.2.1.2).

As mentioned above, the Manifest contains information used to identify the Service being requested. The data describes one or more "attributes" associated with the Service. Using these attributes, the client requesting the Service may describe the Service only as explicitly as necessary. For example, a Service may register a Name or token that identifies the API the Service implements. Applications that do not care which implementation of the Service they get may only provide the Service Name in the Manifest. Alternatively, a Service may be registered with additional "attributes" such as vendor name, hardware description etc. The client may

specify these attributes in the Manifest. When requesting Services, attributes used for selection can be omitted, wild-carded, etc.

Finally, the Service Manifest may be used to pass arguments to the Service. These arguments are Service dependent. In the section **4.2.2 User Mode Service** we will discuss some possible use cases that involve Service Manifest arguments.

Ultimately the AAS Service Factory is responsible for loading and instantiating the "local" Service. That is to say, the portion of the Service executable that resides in the OS Host Process space. More will be discussed on that in the section 4.2.2**.**

4.2.1.4 Event Delivery Service

The AAS Event Delivery Service (EDS) is a mandatory core service. EDS is used for delivering AAL Events to objects within the OS Host Process. The most common case is between a Service and client. EDS provides functions for queuing AAL Events[5] and dispatching them to recipients via callback handlers.

AAL Events are formalized objects that implement an AAL-defined interface. AAL defines two basic classes of Event: **Unsolicited** and **Transaction Events.** Subclasses of those Events are the Exception variants. The next sections describe the types of AAL Events in more detail.

4.2.1.4.1 Unsolicited Events

Unsolicited Events are actually the base class of all other AAL Events. Unsolicited Events, more commonly referred to as just Events, are used for notifications where no client request (i.e., Transaction) has been issued. Unsolicited Events provide members for accessing:

- Object that generated the Event
- Identifier of the Event
- The Event's Interface(s)[6]

4.2.1.4.2 Exception Events

Exception Events extend Unsolicited Event. Like Unsolicited Events, Exception Events are not associated with a client request. Unlike Unsolicited Events, Exception Events are used to notify the client that an Exception (i.e., a failure condition) has occurred.

[5] And functors. Functors are described later in the document.

[6] Objects and accessing interfaces will be discussed in later sections.

In addition to the accessors defined by Unsolicited Events, Exception Events provide members for accessing:

- Error Code
- Sub-error Code
- Description String

4.2.1.4.3 Transaction Events

In section 3.4 we defined a Transaction as an active asynchronous operation, executed by a Service as a result of a request from a client. We also described that Transactions undergo state transitions that must be reported to the client. The minimal state transition is from Active to Complete.

Transaction Events are used to carry Transaction state transition notifications. Transaction Event extends Unsolicited Event. In addition to information provided by Unsolicited Events, Transaction Events contain an object called the **TransactionID.** The simplest definition of the TransactionID is that it is an object used by the client to associate a Transaction Event with a Transaction. Remembering from section 3.4 that a client may have multiple Transactions that may be active in a Service simultaneously, it is further possible that the client may have multiple instances of the same Transaction running concurrently. The TransactionID is supplied by the client and allows it to distinguish and appropriately process Transaction Events based on the instance of the Transaction with which they are associated.

TransactionID's are actually a powerful construct in EDS and will be discussed in more detail later in this chapter and further in this document.

In addition to the accessors defined by Unsolicited Events, Transaction Events provide members for accessing:

- The TransactionID object associated with the Transaction instance.

4.2.1.4.4 Exception Transaction Events

Exception Transaction Events extend Transaction Event and are used to indicate an exception has occurred during the operation of a Transaction. Exception Transaction Events add the same additional accessors as the Exception Event.

4.2.1.4.5 Extending Events

Any Service may extend the basic AAL Event classes by adding its own interface to the Event object.

4.2.1.4.6 Functors

A functor is a type of object that can be used to execute a function call, typically as a deferred operation. The AAL Event class and EDS define a Functor variant that can be used to allow an arbitrary function to be executed by the Event object as opposed to the object being dispatched to an event handler for processing. This is used to enable a service to schedule functionality to be performed.

4.2.1.4.7 Event Dispatching

When an Event is queued for delivery with EDS, the desired destination is specified. EDS later dequeues the Event and dispatches it to the desired recipient via a call to the recipient's callback handler. Generally the callback is provided to the object generating the Event, usually a Service, explicitly. For example, by convention a default callback handler is provided to all Services when they are created. Unless otherwise arranged, this would be the callback used when generating Events. EDS does support a mechanism that allows clients to specify and "override" the default handler.

4.2.1.4.8 Concurrency model/Thread abstraction

EDS defines a thread abstraction to the client. When EDS delivers an Event to a recipient's callback handler, it does so in its (EDS's) own thread context. As a result, EDS defines the concurrency model the application runs in. The concurrency model defines how Events are scheduled. **Event Handlers should NOT make assumptions regarding the OS thread context in which they are running.**

EDS defines three concurrency models:

- Non-preemptive
- Serialized
- Reentrant

Each of these will be discussed in the following sections.

4.2.1.4.8.1 Non-preemptive

In this model, EDS will only deliver Events one at a time, when the process yields a thread to the system using the **EventDeliveryService::Schedule()** call. The Schedule() call will use the applications thread of control to deliver the next available Event. The Schedule() call is a blocking function and does not return until an event has been dispatched.

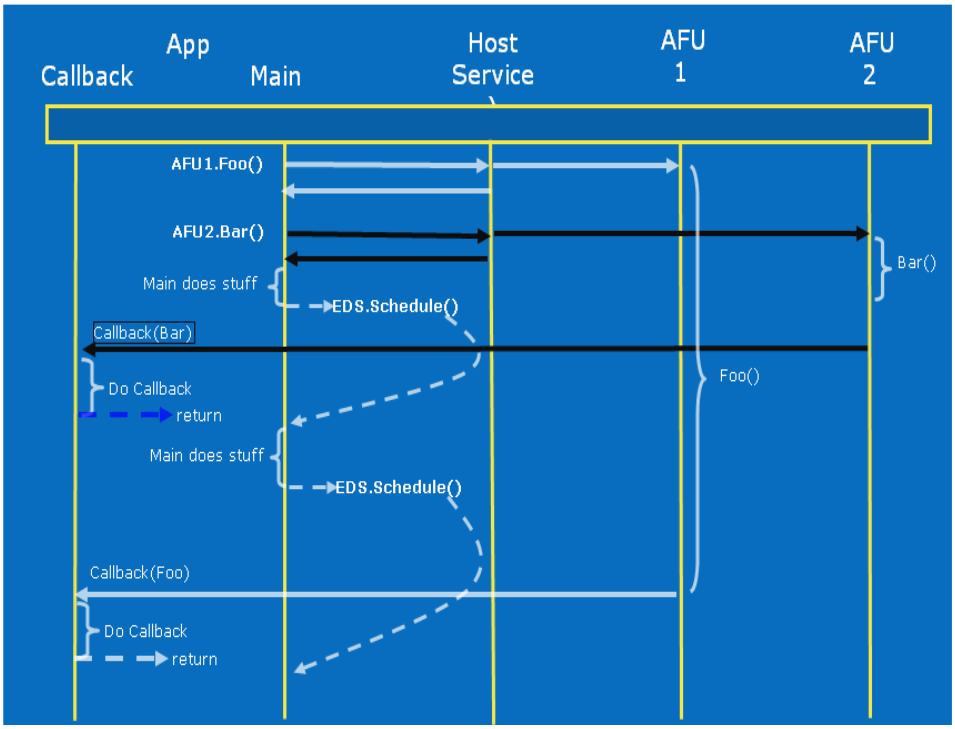

Figure 13 – Non-preemptive concurrency model

Figure 13 illustrates the flow of the non-preemptive concurrency mode. In the example, two services are involved, AFU1 and AFU2. The Application asynchronously invokes methods AFU1.Foo() and AFU2.Bar() causing the execution of these method to execute in parallel. While Foo and Bar execute, the application continues its own thread of execution. Note that the Bar operation completes before the application is ready to handle the Bar Event. The Event has been queued by AFU2 but not yet dispatched. When the application has become ready to process Events it yields it thread of control by calling EDS::Schedule(). The call to EDS::Schedule() informs EDS that it is OK to deliver an event. Again NOTE that as stated earlier, only a single event will be delivered, even if multiple Events have been queued. This model gives the application maximal control over the concurrency of event delivery. The application is assured that Event Handlers will not be reentered. This model is similar to a polling model. Although this model approximates a single-threaded programming model, the application should not make assumptions about the thread context in which the Event Handler is running.

4.2.1.4.8.2 Serialized

In the Serialized concurrency model, EDS will deliver events to Event Handlers as they are available.

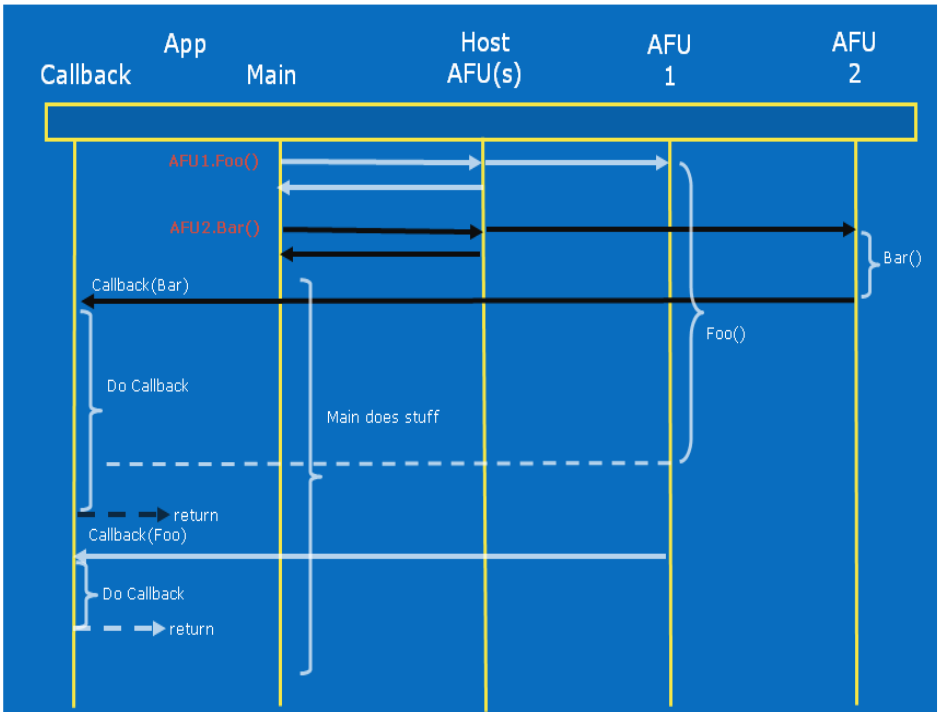

Figure 14 – Serialized concurrency

The Serialized model is named as such because EDS will serialize Event delivery to a **particular Event Handler**. That is, the Event Handler will only process one Event at a time and is guaranteed not to be reentered by EDS. Note that this serialization is assured at the Event Handler and not system or process wide. It is possible but undefined whether other Event Handlers within a process may be invoked concurrently.

The Serialized model provides more parallelism by allowing the application thread to run while Events are delivered. It also simplifies the Event Handler design by ensuring only a single Event will be delivered at a time. The developer need not worry about protecting the handler from reentrancy (from EDS). This model does, however, expose latencies in Event processing for a particular Event Handler based on the amount of time that handler takes to process the event. Note that the Event Foo, in the example in Figure 14, does not get delivered for processing until the handling of Bar is complete.

4.2.1.4.8.3 Reentrant

The reentrant concurrency model provides for the maximum theoretical amount of parallelism by allowing an Event Handler to be processing multiple events simultaneously. In contrast to the Serialized model, where AAL ensures that only a single thread of execution will ever be running in a particular event handler callback at a time, in the reentrant model AAL will allow an event handler callback to be reentered by another thread. The reentrant model requires that AAL be run with EDS thread pooling enabled.

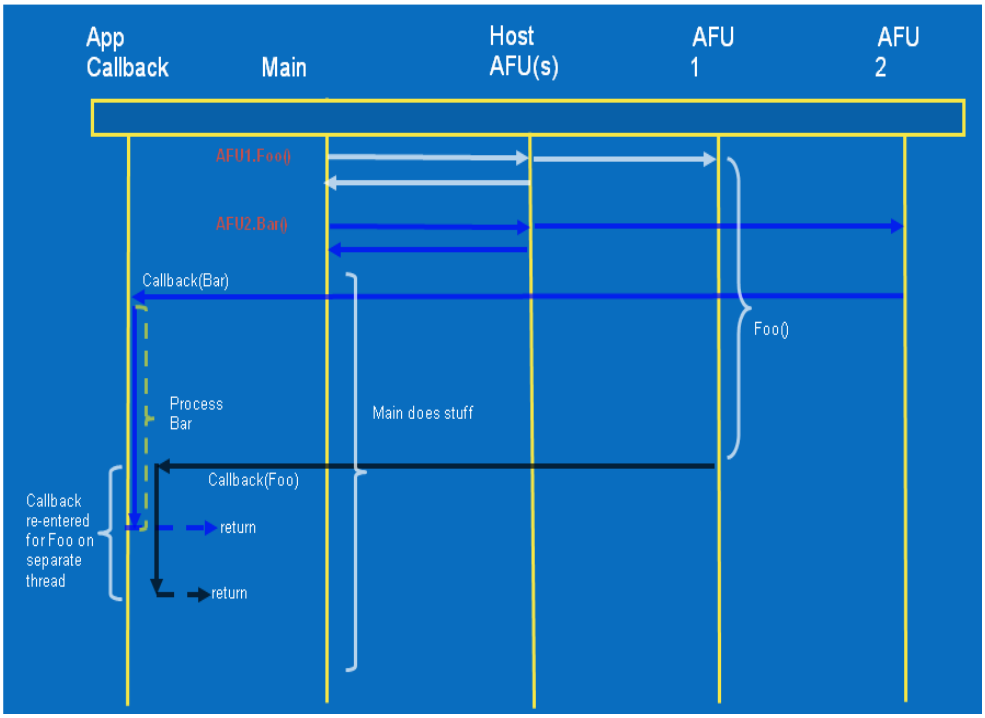

Figure 15 – Reentrant concurrency

Figure 15 shows an example of the reentrant model. Using EDS, AFU 2 sends an Event Bar to the Event Handler Callback(). At some later time, AFU 1 sends an Event Foo to the *same* Event Handler. Assuming EDS is running with a thread pool, this results in two threads of execution running through the Event Handler simultaneously.

Applications using this mode must be designed as "thread safe" using typical mechanisms such as semaphores and mutexes.

While the EDS is designed to use multiple threads for dispatching events, applications CANNOT make any assumptions about the thread a callback is running on. Application designers should not use mechanisms that rely on the thread, such as "Thread Attached Storage". AAL provides several mechanisms for associating application-specific data needed when processing events. Designers are encouraged to use these supported constructs where possible.

4.2.2 User Mode Services

So far in this chapter we've described the facilities and platform services provided by AAL. As has been said, AAL is a Service-Oriented platform. All of the facilities discussed to this point are in support of the real value of an AAL-based platform, the **Services**. In this section we will discuss AAL Services in the User Mode Domain.

A **User Mode AAL Service** is a formal object that has a well-defined structure and must implement a minimal set of interfaces. AAL User Mode Services may run entirely in the same process space as the client or may execute some functionality remotely. In the extreme case, the portion of the Service residing within the application may be little more than a "Proxy" to the Service. AAL does not dictate how Service functionality is partitioned. The AAL User Mode Service SDK provides tools to facilitate designing and building compliant AAL User Mode Services.

AAL User Mode Services fundamentally follow a **Component** design model. The programming interfaces (APIs) that expose the functionality of the Service are logically separate from the implementation. The object the user "sees" is an **encapsulation** of the implementation of the functionality described by the interface.

Conceptually the Service object can be thought of as a **Container** that encapsulates the implementation of the **Service Component Interface**. The Service Component Interface is the one used to perform the real work of the Service. The **Service Container** implements the **AAL::IBase** interface. Applications use the IBase interface to acquire the desired Service Component Interface. Figure 16 depicts this relationship.

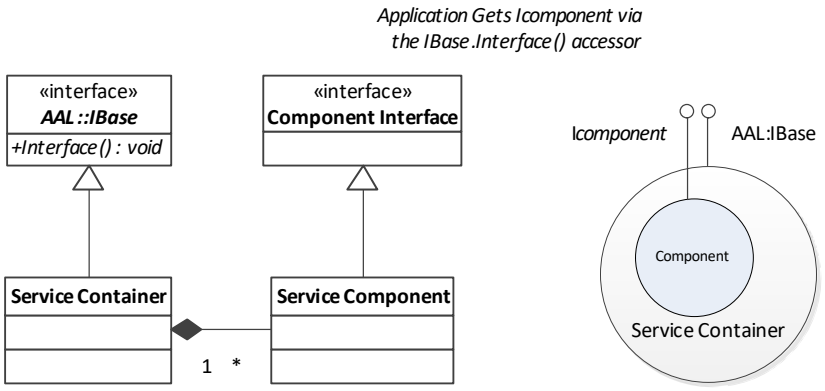

Figure 16 – AAL User Mode Service Container

Clients acquire Services by sending a request to the AAL Service Factory (see 4.2.1.3). If the request completes successfully, the application receives an AAL Event containing a reference[7] to the AAL Service Container object's IBase interface. The client application then uses this interface to get to the desired Service Component API.

This design pattern has a number of advantages over traditional inheritance and is typically a fundamental starting point for Service Oriented Systems. The Component design model lends itself well to Dynamic Software Systems where Services are comprised of Components continually being constructed and destroyed. Loosely-coupled Components and the late binding of interfaces enable the underlying implementations to change without the client application necessarily being aware.

While the model implies aggregation of objects with a Container, AAL does not impose this at the implementation level. It is perfectly acceptable to use traditional inheritance to implement the Service Container using an "is a" relationship. The only requirement is that the Component interface be exposed via the IBase.Interface() accessor.

The AAL C++ SDK provides helper functions and templates that allow the application writer to access the Component Interface as you would a cast operator. Figure 17 shows an example of accessing a hypothetical Service Interface ***IComponentSample*** using one of the AAL SDK casting templates **subclass_ref<>**. In this example the AAL Event ***theEvent*** holds the Service Container object. The **Object() accessor** method returns the IBase interface of the Service Container. The casting template then uses the IBase to return the IComponentSample (reference in this case). The application then simply uses the Component Interface as it would any other object. For more information on the casting templates refer to the AAL SDK User's reference documentation.

```
// Use the AAL casting template to get the IComponentSample interface from the
//    Service Container Object's IBase returned in the the AAL Event object "theEvent"

IComponentSample &rService = subclass_ref<IComponentSample>(theEvent.Object());
rService.DoMyFunction(…);
```

Figure 17 – Using the AAL casting templates

4.2.2.1 Taxonomy of a Service

Now that the basic design concepts of a User Mode Service have been described, we will discuss the structure of an AAL User Mode Service in more detail.

[7] The term "*reference*" here literally refers to a C++ object reference.

User Mode Services are packaged as libraries and are typically dynamically loadable, such as shared objects in Linux or dynamically-linked libraries in Windows. Within the library package the AAL Service implementation consists of at least four components:

- Package
- The Service Module
- Concrete Factory
- Service Implementation (i.e., Service Container)

Each of these will be discussed in more detail in the following sections.

In Figure 18 the outer box represents the shared library or DLL package containing the Service implementation.

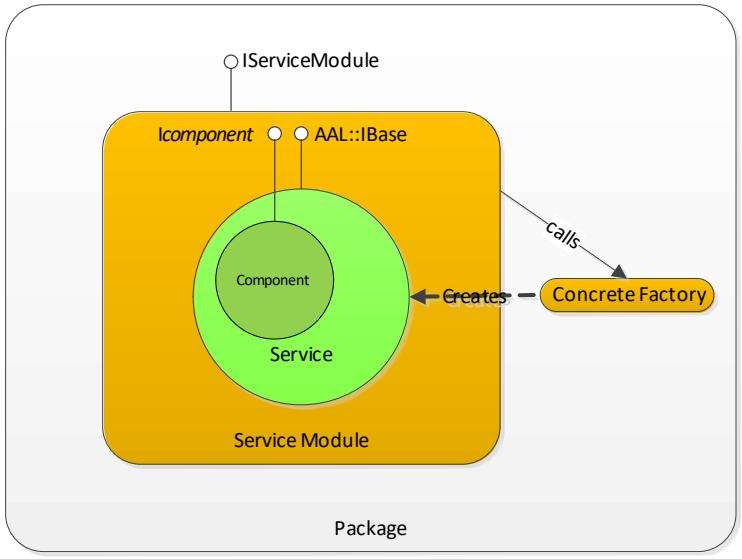

Figure 18 – Taxonomy of a User Mode Service

4.2.2.1.1 Package

The **Package** is the executable environment in which the Service is implemented. The Package is specific to the OS environment. In most cases the Package is a shared library or dynamic linked library but could also be a process or a loadable kernel module. In all cases the Service provides a "**local**" implementation that represents the interface to the service. The "local" implementation may be as simple as a proxy to the Service implementation or may contain the entire Service implementation. The notion of "local" and "remote" Service implementations will be described in more detail in section 4.8.

4.2.2.1.2 Service Module

The Service Module serves an administrative role between the AAL Service Factory and the Service. There is exactly one Service Module instance per Package. When a Service is requested by a client application, the AAL Service Factory first determines if the package containing the Service implementation has been already loaded into the process address space. If so, there is no need to reload the library package. The AAL Service Factory keeps a cache of Service Module objects that represent the library packages implementing each Service loaded. The AAL Service Factory interacts with the Service Module to instantiate the Service implementation. Service implementations can be passed arbitrary input data and parameters from the client application at initialization time via a data object called a **NamedValueSet.** The **NamedValueSet** object is a **map** data structure that uses keys and associated values to store data. NamedValueSets essentially replace C++ structures for passing structured data between objects (See 4.4.1).

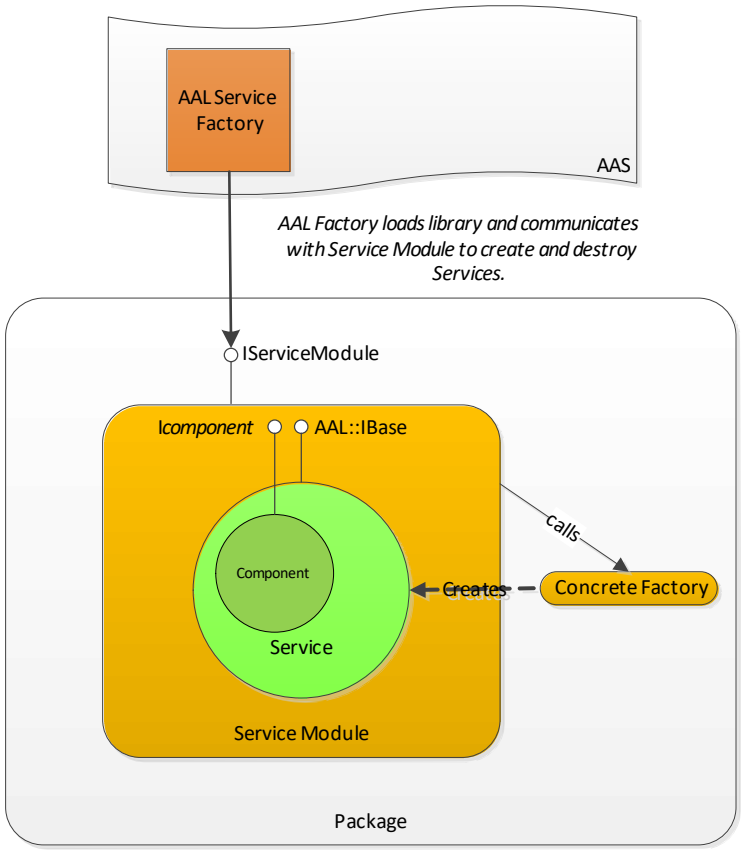

Figure 19 – AAL Service Factory using Service Module

A single Service Module is capable of creating multiple instances of the Service it manages (See Figure 20). It is the responsibility of the Service Module to maintain a list of all of the Service objects that are created through it. In the case of a failure or a "dirty" system shutdown, the

AAL Service Factory may instruct the Service Module to shut down. The Service Module is responsible for shutting down any outstanding Services it "owns" and releasing their resources.

The implementation of the Service Module is generic and is provided by the AAL User Mode Service SDK.

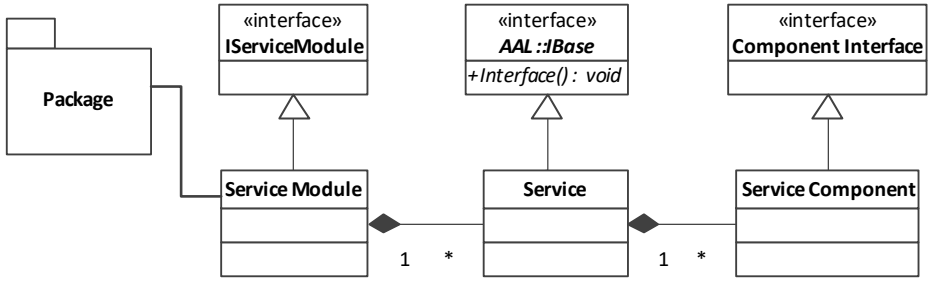

Figure 20 – User Mode Service Package

4.2.2.1.3 Concrete Factory

AAL implements an **Abstract Factory** Design Pattern. The AAS Service Factory provides an interface which allows the client application to instantiate any type of Service available on the platform. The AAS Service Factory does not instantiate Services directly. As was mentioned in the previous section, the AAS Service Factory loads the library package containing the Service implementation. Then, using a well-known interface to the Service Module, the Factory forwards the creation request to the Service Module. The Service Module, which is itself generic, invokes the **Concrete Factory** which is responsible for the actual instantiation of the Service object.

AAL defines a set of interfaces and the associated protocol followed by the Service Module and the Concrete Factory. This enables the Service Module to be generic while supporting a number of different creational patterns supported by the concrete Factories.

The AAL User Mode Service SDK provides generic implementations for several common creational patterns. The simplest pattern is one which creates a new instance of the Service implementation for each create request. Another, somewhat more sophisticated, pattern is the **Singleton**, wherein the package creates only a single instance of the Service object. The Factory may enforce that subsequent create requests fail, or it may allow sharing of the Singleton using smart pointers and/or other synchronization techniques.

Depending on the pattern, the Factory will instantiate an instance of the appropriate Service object and then call the object's initialization entry point. The remainder of the Service's initialization is performed inside the object itself. This partitioning allows Concrete Factory to be largely generic.

4.2.2.1.4 Service Implementation

The Service implementation is where the real work is performed. At the beginning of this section we described the Service as being composed of a **Service Container** and one or more **Service Components**. While this is certainly the logical organization of a Service, AAL does not impose this structure on the design of the Service.

As was previously discussed, Services must implement the **AAL::IBase** interface. This interface is used to query for additional interfaces implemented within the Service. Interfaces are registered with the Service Container by associating a C++ interface pointer with a unique numeric ID (e.g., GUID). This is similar to other component designs, such as **Microsoft COM** (Component Object Model) technology. The AAL User Mode Service SDK provides the implementation of the Service Container and its associated IBase interface. The Service designer registers their specific interfaces with the Service Container using APIs provided by the SDK. Typically this is done during Service construction or initialization but could be performed at any time. A Service may delay the allocation of **Compute Resources** it needs to perform its work until the last possible moment by not instantiating an interface until it is needed.

Using the AAL User Mode Service SDK, Service developers derive their Service Container class from one of the SDK base classes. The base class provides the implementation for the Container. Additional functionality may be provided by implementing Service Components as discrete class objects, or the designer may choose to simply implement the service functionality directly in the Service Container. In this case the Service still registers its interface with the Container, but it does so by simply registering itself (i.e., the object's this pointer). When all of the Service implementation is within the Service Container, the Container/Component structure is simply logical.

Figure 21 shows an example Service implementation where the Service implements one interface through inheritance (Interface 1) and another through aggregation (Interface 2).

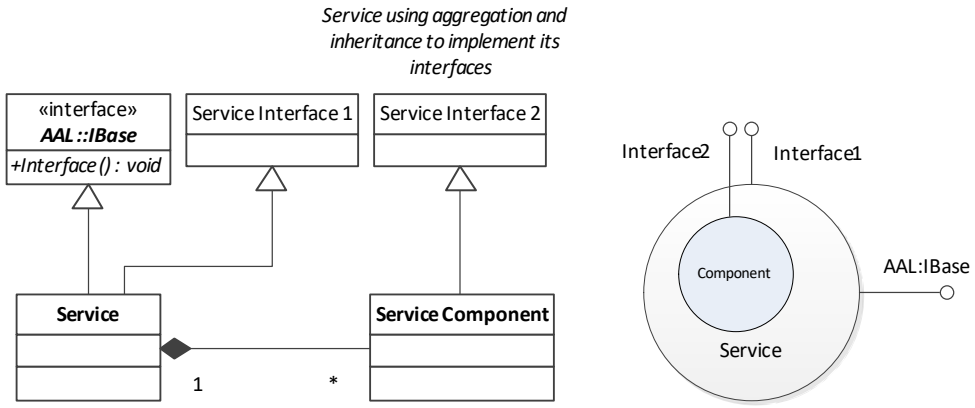

Figure 21 – Service Implementations Using Aggregation and Inheritance

The AAL User Mode Service SDK provides C++ classes, templates, and interfaces to facilitate the construction of compliant AAL User Mode Services. To insure a reliable and deterministic start-up and shutdown behavior, the SDK specifies certain interfaces that must be implemented by the Service and certain minimal guidelines the Service designer must follow. By following these guidelines the developer will be guaranteed that their Services and components will interoperate reliably with other Services and components in the system.

In section 4.2.2.1.3 we described how the **Concrete Factory** is responsible for the instantiation of the Service implementation. In order to keep the Concrete Factory as generic as possible, the Factory's role in Service construction is limited to the instantiation of the bare object (e.g., C++ *new*) and then passing the newly-created object the initialization parameters supplied in the **NamedValueSet** to the Service Module (See section 4.2.2.1.1).

4.2.2.1.4.1 Observer Model

In this section we will discuss an important aspect of the AAL programming model. AAL Services implement an **Observer** Design Pattern. In the Observer pattern, a **subject** of observation maintains a list of **observers** interested in being notified of state changes on the subject. When a noteworthy event occurs in the subject it generates a notification message to the observer. In this case the subject is the Service and the Observer is the client application.

When a Service is constructed on behalf of a client, the client provides a **callback** function as one of the input arguments to the AAL Service Factory's **Create()** method. This callback is passed down to the Service where it is registered as the default target for notifications from

that Service. AAL requires clients to provide a callback interface when acquiring a Service, regardless of what other mechanisms the Service may define to communicate with the client. This insures that there is **always** a target for notifications. Section 4.2.1.4 discussed the AAL's **Event Delivery Service (EDS)** platform service. EDS provides the facilities for passing events from object to object. Services use EDS to deliver Events to Observers.

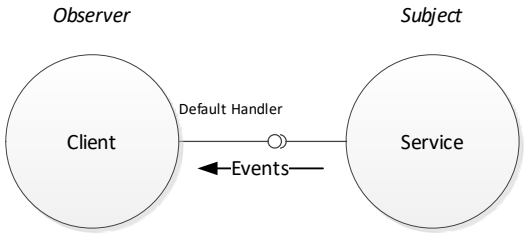

Figure 22 – Observer model

Two notifications that are guaranteed to be generated to the default handler are the Creation Event and the Release Event. The Creation Event is generated by the Service and contains a reference to the Service's AAL::IBase interface. This is the mechanism by which the application acquires the Service. The Release Event is a notification indicating that the Service is no longer available to the client.

4.2.2.1.4.2 Additional Programming Models

While the AAL defines an Observer model as the default method that notifications are passed to the client from the Service, it does not mandate it as the only protocol a client and Service may implement. AAL supports or allows significant flexibility in the design of interfaces between client and Service. In the following sub-sections we will touch upon a few. This is not meant to be an exhaustive list and in some cases a combination of techniques may be implemented.

4.2.2.1.4.2.1 Transaction-Based Callbacks

In section 3.4 we defined a **Transaction** as the asynchronous operation being performed by the Service. In the Observer model, the Events associated with the Transaction will be delivered to the default event handler callback. In the Transaction-based Callback model, Services implement asynchronous methods that take a callback handler as an argument to the method. The Service uses that handler as the target of Events associated with that Transaction. The AAL Event Delivery Service defines an object, called the **TransactionID,** that facilitates implementing Transaction-based event models. A common design pattern is for the Service to define a reference to a TransactionID object as its last argument in the function signature.

Example:

```
void PingOne(AAL::btcString sMessage, AAL::TransactionID &rTranID)
```

The TransactionID was designed to assist in writing asynchronous event driven models and state machines. EDS **Transaction Events** (see section 4.2.1.4.3) contain a copy of the TransactionID object passed when a Transaction was started. TransactionID objects have a number of attributes that can be set when the object is created. These include a user-defined numeric value or ID, a user-defined context (void *) allowing the client to carry Transaction related data, and a pointer to an event handler. When the Service posts a Transaction Event to EDS for delivery, the EDS examines the enclosed TransactionID; and, if an event handler pointer is present, then EDS will deliver the Event to the embedded callback handler. The TransactionID has an attribute that tells EDS whether to ONLY deliver the Event to the embedded handler or to deliver the Event to both the embedded AND the default handler as well. For more information on the TransactionID, see section 4.4.3 the AAL SDK Programmer's Reference.

4.2.2.1.4.2.2 Peer Model

The AAL implementations of the **Observer** and **Transaction** models are both Event-oriented models. That is to say, they rely on messages back to the client being packaged in AAL Event objects. These models also imply an asymmetrical relationship between the objects, as evidenced by the use of the terms "***client***" and "***service***". The relationship is oriented toward the client making requests of the Service and the Service responding. In a client/server model it is unusual for the Service to initiate a Transaction or dialog. Unsolicited Events are generally indicators of a failure on the Service or platform.

The Peer model is oriented to usage models where the communicating objects may not have a clear hierarchy or where the use of Events is cumbersome. In the Peer model the client passes an interface pointer of its own to the Service. The Service may then call methods on the client's interface directly rather than generating an Event.

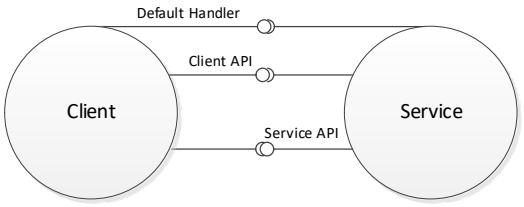

Figure 23 – Peer model

AAL does not define how the Client passes its interface to the Service. This is up to the Service designer to decide; however, there are two basic approaches. The client may provide its interface when the Service is created by including it in the **NamedValueSet** object passed in the initialization parameters of the **AAL::Factory.Create()** function. The Service may also define a custom method, such as a specific member function, for passing in the client interface.

4.2.2.1.4.2.2.1 Concerns with Peer Model

One benefit of the Event-driven model is that the Service need not be concerned about threading and reentrancy with regard to the delivery of the event. The AAL Event Delivery Service takes care of that (see section 4.2.1.4.8). However in the Peer model, the Service and client may run into problems with recursion if precautions are not taken. Consider the scenario where the Client calls a method in the Service and that Service method then directly calls an interface in the client as a response. The client then directly calls from that method another Service method… and so on. The system will likely soon fail with a stack overflow. There are a number of solutions to this problem. Fortunately AAL, specifically the Event Delivery Service, provides a mechanism for solving this problem. EDS defines a special type of Event object called a **functor** (see section 4.2.1.4.6). Functor Events allow the user to define an Event that contains an object or interface pointer. When EDS "dispatches" the Event it results in the Event invoking the desired interface in the EDS event delivery thread context. For more information on functor Events see the AAL SDK Programmer's Reference.

4.2.2.1.4.3 Nested Services

A powerful attribute of a Service-Oriented Architecture such as AAL is the ability to easily combine (i.e., nest) Services to construct increasingly sophisticated Services while maximizing reuse. A Service implementation may use other Services (as a client of that Service) to perform its work. It requests this **subordinate Service** through the AAL Service Factory, exactly like any other application client.

A Service may use a subordinate Service any way that it wishes to perform its work. A simple example of how one Service may use another is if a Service wanted to aggregate the functionality of another Service, exposing the subordinate Service interface directly to its client. It could do so by simply requesting that Service and then registering the Service's interface with its own Container. The client application calls the subordinate Service's interface directly, having acquired it through the superior Service's IBase interface.

4.3 Client Application

In a traditional client/server model the distinction of "**client application**" is common. The application is an executable, often implemented in an OS process, which performs some work on behalf of a user. As the name implies, the relationship the client application has to the platform resources is that of a consumer. In a purely Service Oriented framework the distinction between client and service can be more obscure. In a Service Oriented system such as AAL, an executable can be a client of Services while being a Service to other clients.

Nevertheless, most systems will contain objects that perform a purely consumer role and as such are considered client applications.

There are a couple of primary characteristics of AAL client applications:

- They exist within an OS process (refer to Figure 8).
- The process must include the AAL core library and have initialized it using **AAL::SystemInit().**
- There must exist a system level event handler.
- The process must call **AAL::SystemStop()** when complete.

Much like with AAL Services, the AAL core communicates with the client application through AAL Events. The client application supplies a system event handler at initialization time. The system event handler is used by the AAL core to dispatch system notifications. The first system Event the client application receives is a "**System Initialized"** Transaction Event (See 4.2.1.4.3 for more on Transaction Events). The "System Initialized" Event contains a reference to a **ServiceContainer** object. The ServiceContainer has methods that provide the interfaces to the AAL core services, such as the AAL Service Factory. The application should call the AAL::SystemStop() function to shutdown the AAS Core services. Only a single instance of the AAL core is present per OS process.

4.4 AAL User Mode Domain API Data Objects

AAL provides a number of C++ objects that are used in the framework. **Named Value Sets** are general purpose data containers used to carry information around the framework. **TransactionIDs** are objects defined by the Event Delivery Service that allow asynchronous calls and their returned messages to be connected and nested, leading to clean state machines.

4.4.1

4.4.2 NamedValueSet

Named Value Sets (**NVS**) are the primary mechanism to transport structured data such as arguments around the AAL framework. One would use structs in C, but an NVS is more flexible and the number and type of arguments passed is extensible without recompilation.

A Named Value Pair is a Name, and a Typed Value. For example, a name of "Argument 1" associated with a Value of type Int32 and a value of 65000.

A Set of Named Value Pairs is a Named Value Set. In an NVS, no two Named Value Pairs can have the same Name. Alternatively stated, all Names within an NVS must be unique.

Names can either be strings, e.g. "Executable Name", or "Argument 1", or an unsigned 64-bit integer, or an AAL_ID_t.

The Basic Types that are currently storable in a Named Value Set known in AAS are defined in `include\AALTypes.h`:

Type Name	Type Description
`btBool`	Boolean, TRUE or FALSE
`btByte`	8-bit signed
`btInt`	Native size signed integer
`btUnsignedInt`	Native size unsigned integer
`bt32bitInt`	32-bit signed
`btUnsigned32bitInt`	32-bit unsigned
`bt64bitInt`	64-bit signed
`btUnsigned64bitInt`	64-bit unsigned
`btFloat`	Single precision float
`btString`	ASCIIZ String, (e.g. null terminated)
`btObjectType`	Pointer to any object (e.g. a void*)
`btNamedValueSet`	Recursive – the contents are a NVS
`btByteArray`	Array of btByte (e.g. a binary blob)
`bt32bitIntArray`	Array of 32-bit signed
`btUnsigned32bitIntArray`	Array of 32-bit unsigned
`bt64bitIntArray`	Array of 64-bit signed
`btUnsigned64bitIntArray`	Array of 64-bit unsigned
`btFloatArray`	Array of single-precision float
`btStringArray`	Array of btString
`btObjectArray`	Array of btObjectType

Table 1: Named Value Set Types (Enum eBasicTypes)

Named Value Sets are created like any normal C++ object by declaration or allocation. Enclosed Name Value Pairs can be added, removed, updated, queried, and enumerated.

Their type is NamedValueSet. For example, to create a Named Value Set and add the Named Value Pair "Executable":".\AFU-Vendor1-Stream3Algm.dll" to it, one would execute the code fragment:

```
NamedValueSet nvs;
nvs.Add ( "Executable", ".\\AFU-Vendor1-Stream3Algm.dll" );
```

Example 1: NamedValueSet::Add()

The `btcString` type is available for cases where strings should be constant, such as when retrieving a String from a NamedValueSet or providing a String Name for one.

All operations on a NamedValueSet return an `ENamedValues`, defined in `include\AALNameValueSet.h`.

```
//ENamedValues - Return codes for CNamedValues methods
typedef enum ENamedValues {
   ENamedValuesOK=0,
   ENamedValuesNameNotFound,
   ENamedValuesDuplicateName,
   ENamedValuesBadType,
   ENamedValuesNotSupported,
   ENamedValuesIndexOutOfRange,
   ENamedValuesRecursiveAdd,
   ENamedValuesInternalError_InvalidNameFormat,
   ENamedValuesInternalError_UnexpectedEndOfFile,
   ENamedValuesOutOfMemory,
   ENamedValuesEndOfFile
} ENamedValues;
```

Example 2: Declaration of ENamedValues

In addition, there are various helper functions for Named Value Sets that are not member functions. These are primarily for serializing and de-serializing them to and from flat buffers.

Stream operations are supported on NamedValueSets, so for example to see an NVS for debugging, just write it to a stream, e.g.:

```
   NamedValueSet nvs;
   nvs.Add( "name", "data");
   cout << nvs;
```

Example 3: Stream output of NamedValueSet

For serializing to and from a buffer, refer to the extensive documentation in the headers in the code, specifically the functions `BufFromString()` and `NamedValueSetFromCharString()` in "`include\AASNamedValueSet.h`".

For serializing to and from a `std::string`, refer to the extensive documentation in the headers in the code, specifically the functions `StdStringFromNamedValueSet()` and `NamedValueSetFromStdString()` in "`include\AASNamedValueSet.h`".

The function `NVSMerge()` is available to merge two NamedValueSets.

The stream serialization functions are also available for File I/O, but those may be deprecated in the future. To find those functions look in "`include\AALNamedValueSet.h`" under `NVSReadNVS()` and `NVSWriteNVS`. For a detailed description of the Named Value Set APIs refer to the *AAL SDK Programmers reference guide*.

4.4.3 TransactionID

As described in section 4.2.1.4.3, the AAL Event Delivery Service defines TransactionID objects that are used to allow the application to distinguish, and appropriately process, Transaction Events based on the instance of the Transaction they are associated with. TransactionIDs are simple, C++ objects that are directly constructed and copied in the usual manner. Specifically, a TransactionID holds 4 fields:

- **Context**: void* - used to carry context information for the transaction, e.g. a pointer to an object or a structure containing state that is specific to this transaction.
- **ID**: a 32-bit integer that is by default initialized to a unique number, but may be set by the user to any desired value. As a unique number it can be used directly to match a function invocation with its returned event. If set by the user, it can represent any useful tidbit of information, e.g. an array index indicating which buffer is being manipulated.
- **EventHandler**: a pointer to an Event Handler Function. If non-NULL, this function will be called by the Transaction Method.
- **Filter**: a Boolean, which if true, "filters out" the call to the Default Event Handler after the call to the EventHandler in the TransactionID is made. Filter is only meaningful if EventHandler is non-NULL. If false, then no filtering is performed; that is, both the Event Handler in the TransactionID (if non-NULL) is called, and then the default Event Handler is called.

There are many variants of Constructor. After construction, the individual fields can also be queried and set by their respective accessors and mutators. For a detailed description of the TransactionID APIs refer to the *AAL SDK Programmers reference guide*.

4.5 Kernel Mode Service Domain

The **Kernel Mode Service Domain** (KMSD) executes in a higher privilege mode of the processor typically reserved for the OS and device driver services. KMSD implementations are much more Operating System specific than UMSD implementations due to the privilege level. However even the KMSD provides common architectural elements that transcend OS specifics. This section will discuss the architectural elements common to AAL KMSD implementations.

The KMSD provides several platform services, but its primary function is to provide access to Services executing on remote hosts such as programmable accelerators. In section 4.1 we introduced the notion of **Domain Bridges** that implement protocols allowing access to Services from different domains. The Domain Bridge serves two purposes in the system. It provides a mechanism by which a Service residing on a remote platform or on a different Domain can register with the Resource Management system and Registrar. It also provides an abstract transport mechanism through which client and Service communicate.

Generally AAL configurations fall into two types: Remote Services on a single domain and Remote Services in shared domains. We define **Remote Services** as those that require a Domain Bridge for inter-process, inter-processor or inter-node communications between the client and the Service. The most common example of a Remote Service is one where the algorithm is executing on an accelerator. This is similar to a traditional application controlling a device through a device driver interface but **it is not the same**. In section 4.2 it was mentioned that User Mode Services may utilize hardware accelerators directly using proprietary methods such as accessing the hardware through pre-existing device drivers and libraries. In this case the hardware based implementation of the Service is not visible to AAL and as such is undefined. From an AAL point of view this type of configuration is still a User Mode Domain Service. The hardware resources may not be shared or reconfigured by AAL.

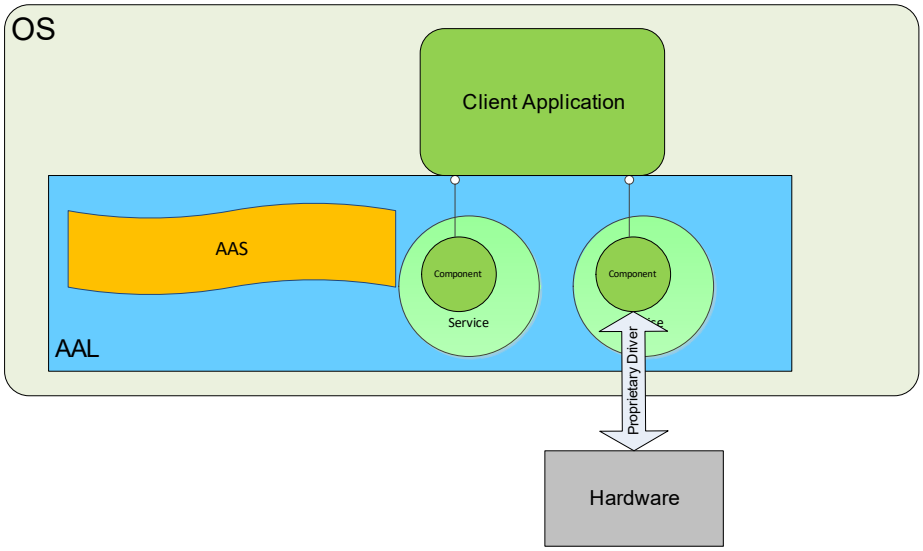

Figure 24 – Using Hardware with Proprietary Interface

Figure 24 depicts this configuration. The hardware is shown outside of the AAL domain.

In contrast, Remote Services register with the Domain Registrar and communicate through an AAL Domain Bridge. This means that a Remote Service is more tightly integrated into the AAL platform services than a traditional standalone driver example would be.

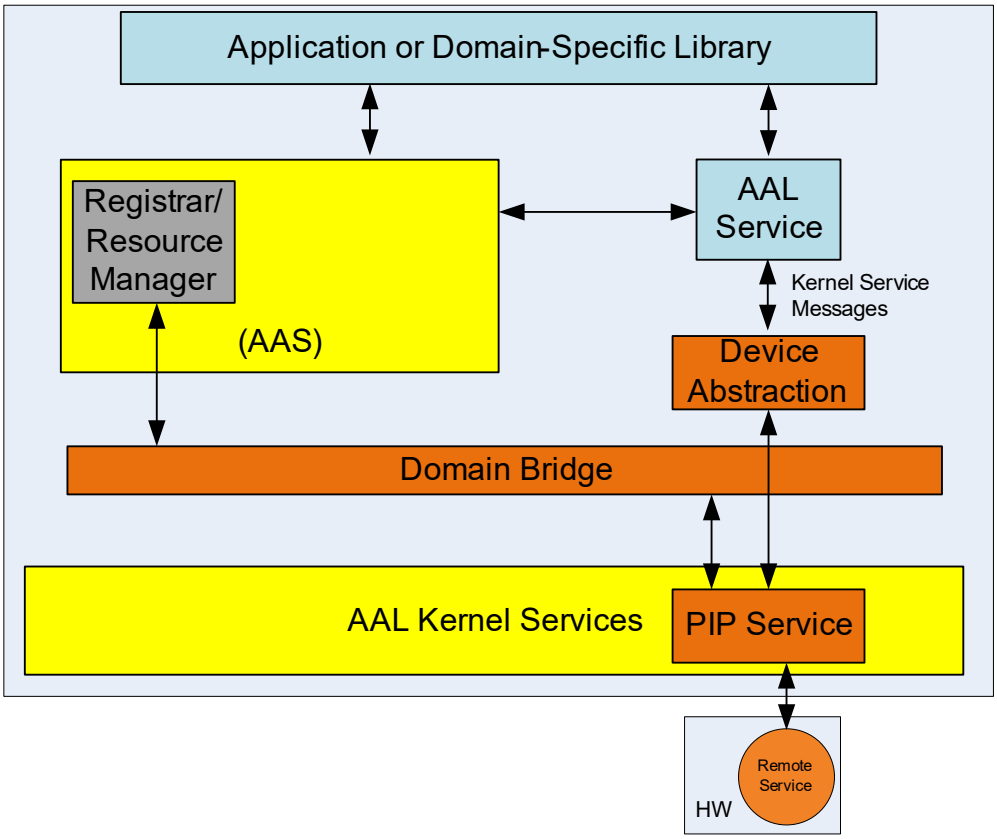

Figure 25 – AAL Kernel Mode Domain Components

Figure 25 depicts the major components of the KMSD. The 4 blocks in orange, the Device Abstraction, Domain Bridge, PIP Service and Remote Service will be covered in more detail in this section.

4.5.1 AAL Kernel Services

The AAL Kernel Services (**AKS**) serve the same function in the KMSD as the AAL Abstraction Services (AAS) do in the User Mode Service Domain. The core platform services for the domain are implemented in the AKS. The Kernel Mode Service Domain is based on a Service Oriented Architecture and as such includes the basic components and functionality of a SOA described in section 3.1. The implementation details of the AKS are platform-specific but in all cases there exists a Registrar and one or more Services. The AKS must also implement transport facilities to allow services to communicate. Figure 26 shows an implementation of a Kernel Service framework for the Linux Operating System. The details of the Linux implementation are beyond

the scope of this section, but we will use this example to discuss the common architectural elements.

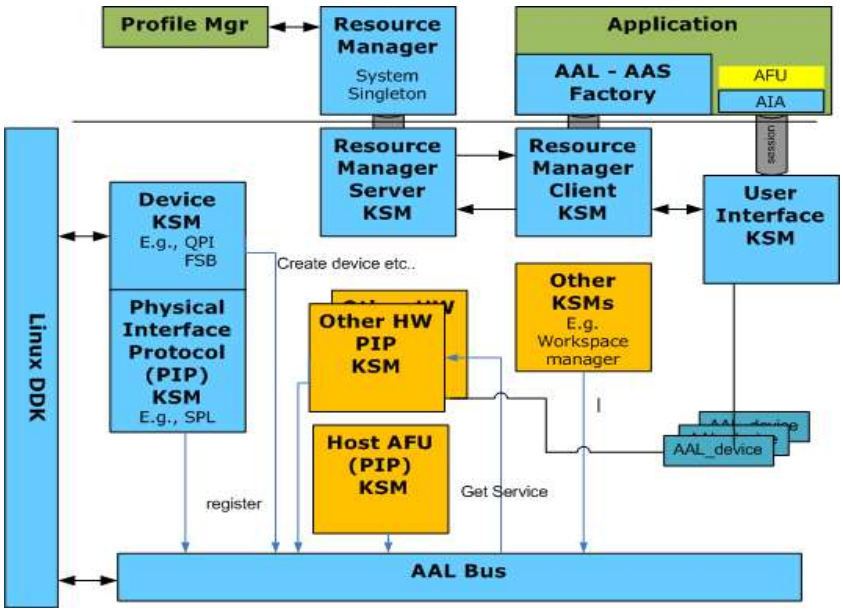

Figure 26 – Example Linux Kernel Framework Architecture

4.5.1.1 Physical Interface Protocol (PIP) and AAL Devices

The Physical Interface Protocol (**PIP**) is a Service that provides the API to the Kernel Mode Service. **AAL Device** represents an instance of an AAL Kernel Service. The AAL Device by itself has no functionality. It must be associated with a PIP to provide its functional interface. AAL defines a base interface for PIPs. The **aal_ipip** interface is primarily a message passing interface. Developers implementing new PIPs may extend the aal_ipip interface if they wish. The aal_ipip interface provides all of the "hooks" necessary to enable the Service implementation to register and communicate with other Services (or clients) in the system.

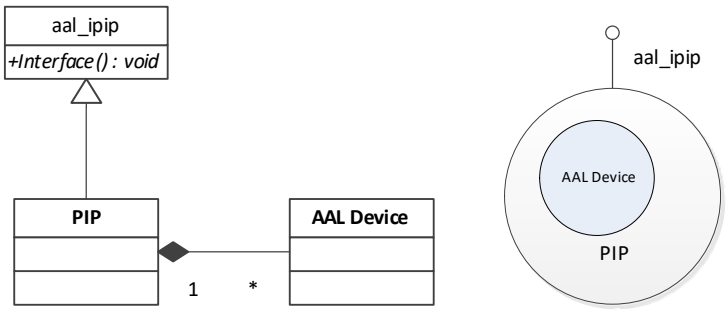

Figure 27 – PIP and AAL Devices

In Figure 27 we show the cardinality of PIP to AAL Device as 1 PIP to many AAL Devices. This is because the PIP provides the interface implementation of the Service. That implementation can

be shared among multiple instances of the same Service. So if the PIP provides the interface implementation then the AAL Device provides the state (i.e., context). The AAL Device could be thought of loosely as the "this" pointer of the Service.

4.6 Software only Kernel Mode Services

As was said, the PIP provides the implementation of the Kernel Mode Service's interface. In the case that the Kernel Mode Service is implemented purely in software then the PIP implements the full behavior of the Service.

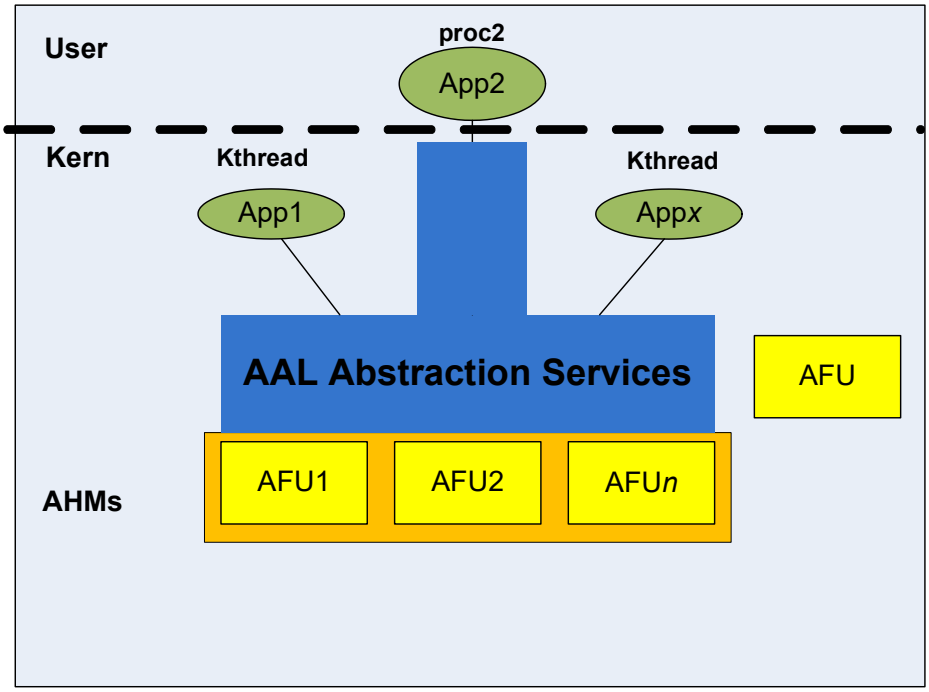

Figure 28 – AAL with Objects in Host Kernel

Figure 28 shows a configuration with clients and Services residing in the host client in kernel mode.

4.7 Hardware Services

More commonly all or part of the Service behavior is implemented on an external device such as a programmable accelerator. The PIP may provide none of the salient behavior of the Service. In this case the PIP provides the functionality found in a traditional device driver. It marshals messages from upstream (the client) and passes them to the hardware algorithm using whatever means is appropriate for the hardware. As with any device driver the PIP would also handle hardware interrupts from the device.

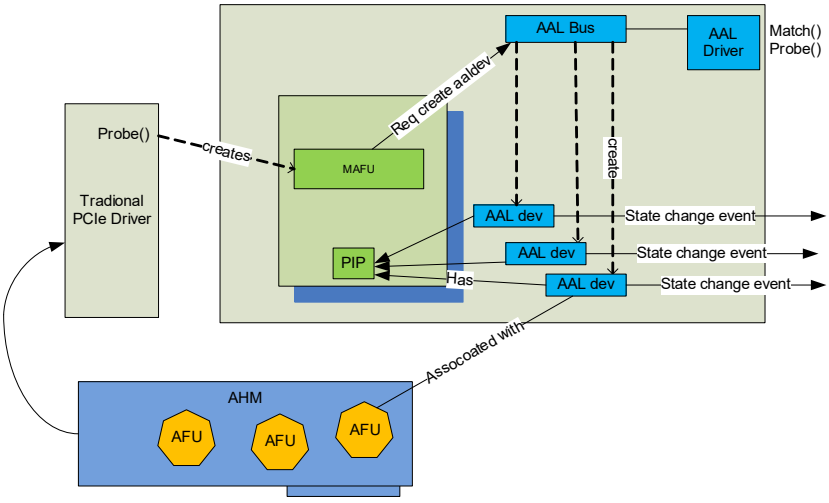

Figure 29 – Example Hardware Based Remote Service Design

Figure 29 shows an example of how a hardware based Kernel Mode Service is designed for Linux.

Note that there often must exist an Operating System specific traditional device driver for the device hosting the Services. The design of that component is beyond the scope of this document.

AAL does not strictly define when and how a Service is instantiated. In practice there are Resource Management Services that control this.

Whether a Service is Software or Hardware based, once it has been instantiated it must register with one or more Registrars in order to be discovered. The way this is accomplished is platform specific but involves using an AAL Kernel Service provided for the purpose.

4.8 Remote Service

Services in the Kernel may be acquired and used by Kernel-based entities just like we've seen in the User Mode Service Domain. All of the same capabilities are available such as aggregation. In

the case that the client of a Service does not reside in the same address space as the Service, the Service is considered to be **Remote**. Note while this discussion appears under the Kernel Service Domain section the distinction is applied universally. For example if a client application uses a User Mode Service that resides in a different process, that Service is considered to be Remote.

4.8.1 Domain Bridge

Once the Service has been registered it must have a Domain Bridge that enables the Remote Service (either SW or HW based) to communicate with its client application.

When the Remote Service is in the Kernel Mode Service Domain, the Domain Bridge will typically consist of some form of kernel/user mode interface such as via a device node in Linux. For example in the Linux implementation there exists a character device driver called the Universal Interface Driver (**uidrv**) and a user mode abstraction library called the Universal Application Interface Adapter (**uAIA**). Together these form the Domain Bridge for User Mode clients to access Kernel Mode Services.

4.8.2 Device Abstraction

The Domain Bridge provides a purpose built transport service between Domains. On the client side of the Domain Bridge there typically exists an object that "represents" the Remote Service in the clients address space. The Device Abstraction is typically little more than a proxy to the Remote Service.

4.9 Summary

In this chapter we looked at the AAL architecture in more detail. The concept of **Service Domain** was introduced as a collection of AAL Services associated with a single **Registrar**. Two examples of a Service Domain from the Linux implementation were described; the **User Mode Service Domain** and the **Kernel Mode Service Domain.** Other Service Domains are possible such as a might exist in a distributed multi-node configuration or where a Registrar exists on an embedded device connected to the host. It is possible for Services to register with multiple Domains.

In section 4.2.2.1 we described the taxonomy of a Service. While the implementation of a Service will be largely affected by the Domain in which it resides, common to all Services is a **Package** containing the implementation; a Service **Module** encapsulating the components; a **Concrete Factory** that can create instances of the Service; and of course the **Service implementation**. Section 4.8 defines the distinction of "local" and "remote" as whether or not a Service's implementation executes within the clients address space. Every Service, regardless of

Domain, provides a "local" implementation of its interface to the client application. Access to a "remote" Service implementation is achieved through a **Domain Bridge.**

Most developers work in the User Mode Service Domain (**UMSD**). The UMSD is the most OS-agnostic Domain. Section 4.2 went into more detail about the implementation of User mode Services and client applications. Platform Services, such as the Registrar, Registrar augmented with Resource Management Services, Service Factory and Event Delivery Service were described in section 4.2.1. In section 4.2.1.4 we explained how the Event Delivery Service (**EDS**) provides intra-process communications services using AAL Event objects. EDS defines the concurrency models and provides facilities for synchronizing Events with the **Transaction** they are associated with. The standard implementation of EDS also provides support for **Functor Events** which enable the deferred execution of a method on an object.

Section 4.2.2.1.4 described the UMSD implementation; that Services are conceptually implemented as **Containers** of **Components**, and how **AAL::IBase** is used to acquire the desired interface. Several supported programming models were described, illustrating the flexibility the developer has over how they construct their application.

In section 4.5 the Kernel Mode Service Domain (**KMSD**) was described. KMSD Services run at a higher privilege level (ring 0). As a result the KMSD is much more OS-specific than the UMSD. While significantly different in design, KMSD shares common attributes with UMSD. KMSD is Service Oriented; it supports dynamic binding of Services; it supports aggregation; and abstracts the Service implementation. In earlier sections it was explained that Services may use hardware based functionality through proprietary interfaces, such as through a traditional device driver. The KMSD enables hardware devices to be more tightly integrated into the AAL platform and take advantage of certain AAL features such as Resource Management. While the primary purpose of KMSD is enabling tight integration of hardware services, section 4.6 described kernel-based Services that are software only. Finally, section 4.7 described how a hardware-based Service can be constructed; how the various AAL components relate to traditional OS specific device drivers and how Kernel Services get registered with a remote Domain (e.g., User Mode).

At this point the reader should have a reasonable understanding of the AAL architectural concepts and be ready to examine an AAL platform implementation.