

A SIMPLE, LIGHTWEIGHT COMMUNICATIONS ARCHITECTURE FACILITATING SCA APPLICATION PORTABILITY

Charles A. Linn (Harris Corporation, Rochester, NY, USA clinn@harris.com)

ABSTRACT

While the Software Communications Architecture (SCA) is a suitable framework for a broad-range of medium and higher capability radio platforms, its use can be non-optimal or precluded on smaller platforms, where software image size, boot time performance or battery life and platform cost become dominant. For these platforms, a lighter-weight solution is desired. On the other hand, the component-based framework and other concepts the SCA provides has proven benefits, and it is desirable to be able to easily port waveforms between an SCA and such lighter-weight frameworks.

In this paper, the “Lightweight Communications Architecture”, or LCA, is described. This framework is appropriate for use on smaller commercial platforms, with land-mobile radio (LMR) systems being a typical example. From a formative standpoint, the design rationale and trade decisions for LCA are detailed, followed by a description of the framework itself. The paper concludes with a discussion on scalability, patterns for facilitating application portability between LCA and the SCA platforms, and performance metrics.

1. INTRODUCTION

When the first version of the Software Communications Architecture (SCA)[1] was released in 2000, a major paradigm shift was made by the US military radio community. Instead of a plethora of closed radio platforms running customized waveform applications, for the first time cross-industry standardized radio platforms running highly portable applications (“waveforms”) started becoming available. While the first platforms were developed under programs funded by the US military, commercially-funded radios also appeared (such as the Harris Falcon™ III series of products), indicating the SCA was a viable and competitive architecture. During the same time, the SCA was breaking out of the military-only community, appearing in applications such as cellular base stations.

While prominent in its target community, the application of the SCA is still relegated to two primary market segments – the military / government market, where

use is essentially mandated, or large-scale radio systems, where the field-upgradability, and often multiprocessor expandability justify the “cost” of the SCA. The vast majority of radio systems are much smaller and simpler, with insufficient size, weight and power (SWAP) to support a CORBA-based SCA. This was the quandary faced by Harris Corporation when we started the design of our Unity™ XG-100 Public Safety Radio product line. On such a radio, battery mission life, fast (less than 3 second) boot time and low cost precluded an SCA solution, but over the past 8 years Harris had internalized the benefits of SCA-based development, and we did not want to regress to the previous “monolithic platform with integrated waveform” second-generation approach that had characterized pre-SCA radios. This same conflict was seen in several other concurrent developments at Harris, making the need clear – needed was a standardized framework for lightweight platforms, much as the SCA has become our standard solution for our military and more complex architectures. This need eventually coalesced into the Lightweight Communications Architecture, or LCA.

2. LCA DESIGN PHILOSOPHY

In creating the LCA, the solution space was narrowed based on several key objectives:

1. *Scale for small platforms:* The SCA already provides a good solution for multiprocessor solutions with requirements for multiple address spaces. If a solution was specialized towards single process / single processor platforms, and by standardizing on C++ as a development language, inter-component calls can be accomplished with simple C++ method calls, eliminating the need for CORBA middleware. This in turn also greatly reduces the memory footprint of the solution.
2. *Maximize portability to / from the SCA at the application level:* To facilitate porting between an LCA and an SCA platform, the structure / form of a *Resource*-based component was maintained to the extent possible. Additionally, the POSIX-AEP was retained. Together, if a “ports by composition” pattern is maintained in conjunction with a *BaseResource* as part of a development kit, the core component

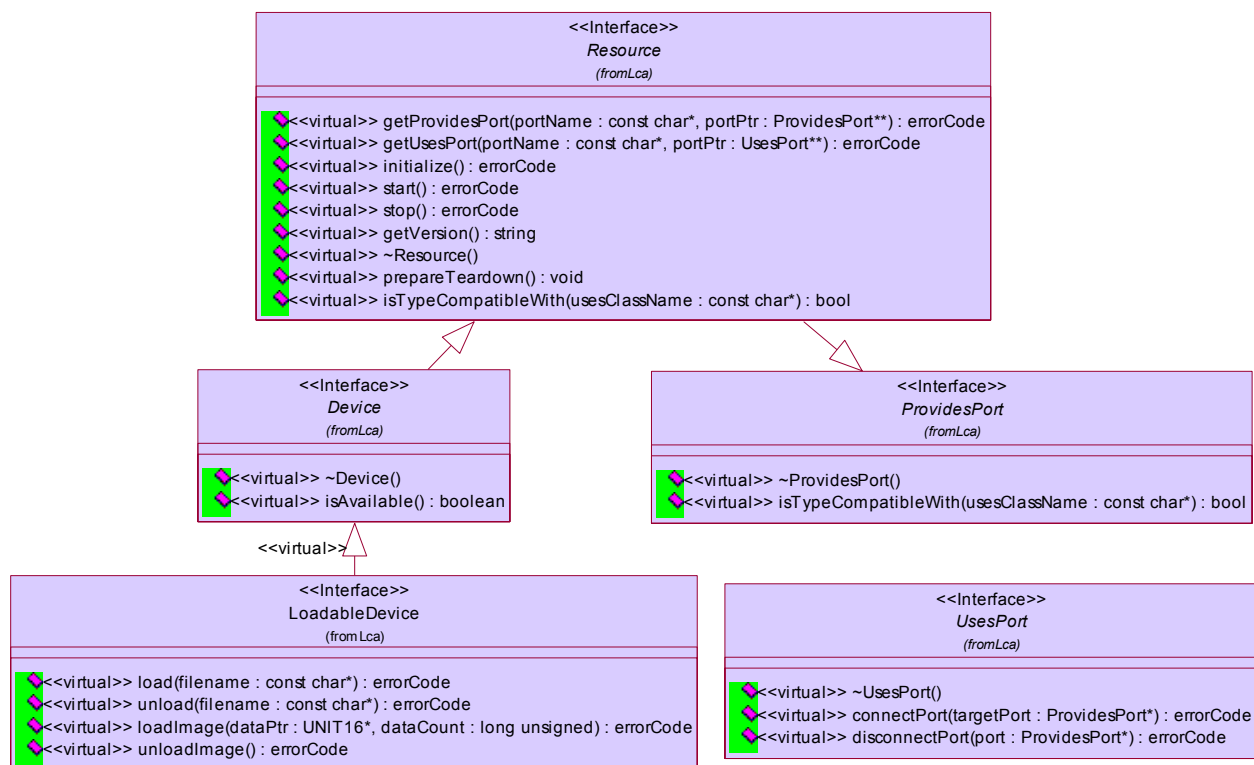


Figure 1: LCA Component and port interfaces

implementation can be substantially ported by substituting LCA port classes and base classes for the SCA equivalents. This is discussed further in section 5.

3. *Pattern after SCA when you can:* The essential SCA architecture at the interface level is well validated, and many organizations are well versed in it. Furthermore, many radio developers are likely to move between LCA and SCA projects. For these reasons, SCA form, names and responsibilities were maintained to the extent practical. Examples of this can be seen in the following sections detailing the architecture.
4. *Maximize implementation flexibility:* Like the SCA, the LCA is specified primarily at the interface level, which allows for variations in implementation including static or dynamic application launch, and pre-linked or separately installable applications.

3. LCA ARCHITECTURE

Like the SCA, the LCA is specified with pure interface classes with specified behaviors and interactions, not modules of executable code. Instead of IDL, C++ pure abstract base classes are used to represent interfaces.

3.1 Component and port interfaces

LCA waveforms and platform software are constructed of components which are interconnected using ports. Each component inherits from a *LCA::Resource*, *LCA::Device*, or *LCA::LoadableDevice* interface supports its operations.

The Resource interface, shown in Figure 1 is modeled after its SCA cousin, with similar semantics for *initialize()*, *start()* and *stop()* operations. The CF *release()* operation was replaced with a *prepareTeardown()*, and supplied with different behavior -- LCA components don't destroy themselves (this is done by the *LCA::ApplicationFactory*), but merely put themselves into a state consistent with teardown. The SCA *getPort()* operation was replaced with a pair of operations, *LCA::getProvidesPort()* and *LCA::getUsesPort()*, to better utilize C++ static type checking. Finally, an *LCA::getVersion()* operation is added to enable standardized queries of component versions.

Notably lacking in the *LCA::Resource* interface is a standardized configure interface. In the SCA this was provided using the CORBA "any" facility to pass name / value pairs to a resource or application "assemblycontroller" component. This centrally-managed interface allowed for the core framework to supply XML-based initial configuration values, as well as providing the theoretical ability for a "generic" user interface to provide XML-driven configuration of an application. As C++ does

not provide a simple, type-safe mechanism for generic typing, and since customized port-based configuration is viable alternative (indeed, this is the de-facto way of configuring SCA platform devices), this mechanism was not included in LCA. Instead, any component may provide a configuration port supporting whatever custom interface it requires.

Separate “uses” and “provides” ports interfaces were defined to match their intended functions, and to provide a lightweight real-time type checking mechanism. *ProvidesPorts* are connected to *UsesPorts* by pointer, but it is prudent for a *UsesPort* to ensure it has indeed been connected with the correct subclass of *ProvidesPort*. This is done through the polymorphic operation *isTypeCompatible()*, which allows the target port to verify that it is indeed compatible with the type supplied by the caller¹. This mechanism also supports type checking when subclassing of *ProvidesPorts* is employed, allowing use of extension patterns. In this case, a more derived (specialized) port could indicate if they were able to also function as one of the more general interfaces, or if this is to be precluded.

LCA *Devices* are considerably simpler than SCA *Devices*, as LCA does not support the dynamic, runtime deployment determination. As a result, allocation property concepts, *usageState* and *adminState* can be eliminated. The only specialized operation present is *Device::isAvailable()*, which allows a *Device* to state if it is operational and hence available for connection to a waveform. This mechanism accommodates the most common cases of cable-sensing devices, missing underlying hardware, offline status, et-cetera. *LoadableDevices*, like in the SCA do not create LCA components (i.e. handle deployment), but are used to load external devices such as DSPs, FPGAs etc. Two standardized methods are available, *Load()*, which loads from a file, and *LoadImage()*, which loads images directly from memory (useful when file-services are limited or slow, with the image data being stored in a C++ header file instead).

LCA does not have an analogy to a *CF::ExecutableDevice*, as it does not support deployment of components on *Devices*. In most cases, the LCA “domain” is on a single processor, and so *Devices* and service components (in LCA a service is simply an *LCA::Resource*) are created by the *NodeManager*, and application *Resources* are created directly by the *ApplicationFactory* components.

3.2 Management interfaces

Figure 2 shows the *NodeManager* and *ApplicationFactory* interfaces, and their relationships to the component classes.

In the LCA, the *NodeManager* is roughly equivalent to a combination of the SCA *DeviceManager*, *DomainManager* and CORBA naming service. Since the LCA does not support multiple processors within a domain (*LoadableDevices* such as DSPs and FPGAs put aside), the need for independently-registering *DeviceManagers* was lost, and as a result the design could be simplified. The *NodeManager* has the following major responsibilities:

1. Creation and interconnection of the platform *Devices* and services, typically at platform boot. As each *Device / Resource* is created, its name and pointer is added to an internal lookup table (the equivalent of the SCA *Device* registration process).
2. Resolving *Device* instance names (each *Device* instance is given a name at creation, and this name is used by the *ApplicationFactory* to connect interested application components) into a *Resource* pointer.
3. Creation of *ApplicationFactory* instances.
4. If supported (optional capability), accept installations of applications, adding the new application to the “create application on *NodeManager* boot” list.

ApplicationFactory components are responsible for the creation of application instances. Unlike in the SCA, where a *CF::Application* instance is used to manage the created application, in the LCA the *ApplicationFactory* itself is responsible for waveform creation, reference counting and teardown. Multiple creation of the same application using unique names is supported, and the *ApplicationFactory* may also be queried to provide pointers to the specified instance.

The created “application” in actuality consists of one or more components (*LCA::Resource*), with a designated assembly controller. These components are connected to each other as well as the platform *Devices* and platform *Resources*, with a pointer to the designated assembly controller being returned to the caller. Each application’s assembly controller is responsible for delegating overall application configuration and control to the individual application’s components, in a manner similar to the SCA.

4. LCA VARIATIONS

As befits a framework targeted for minimalist platforms, considerable versatility is allowed to LCA implementations to accommodate varying requirements and SWAP considerations. This section details some of the more common variations.

¹ C++ RTTI was not used for this purpose as many platforms choose to disable this feature.

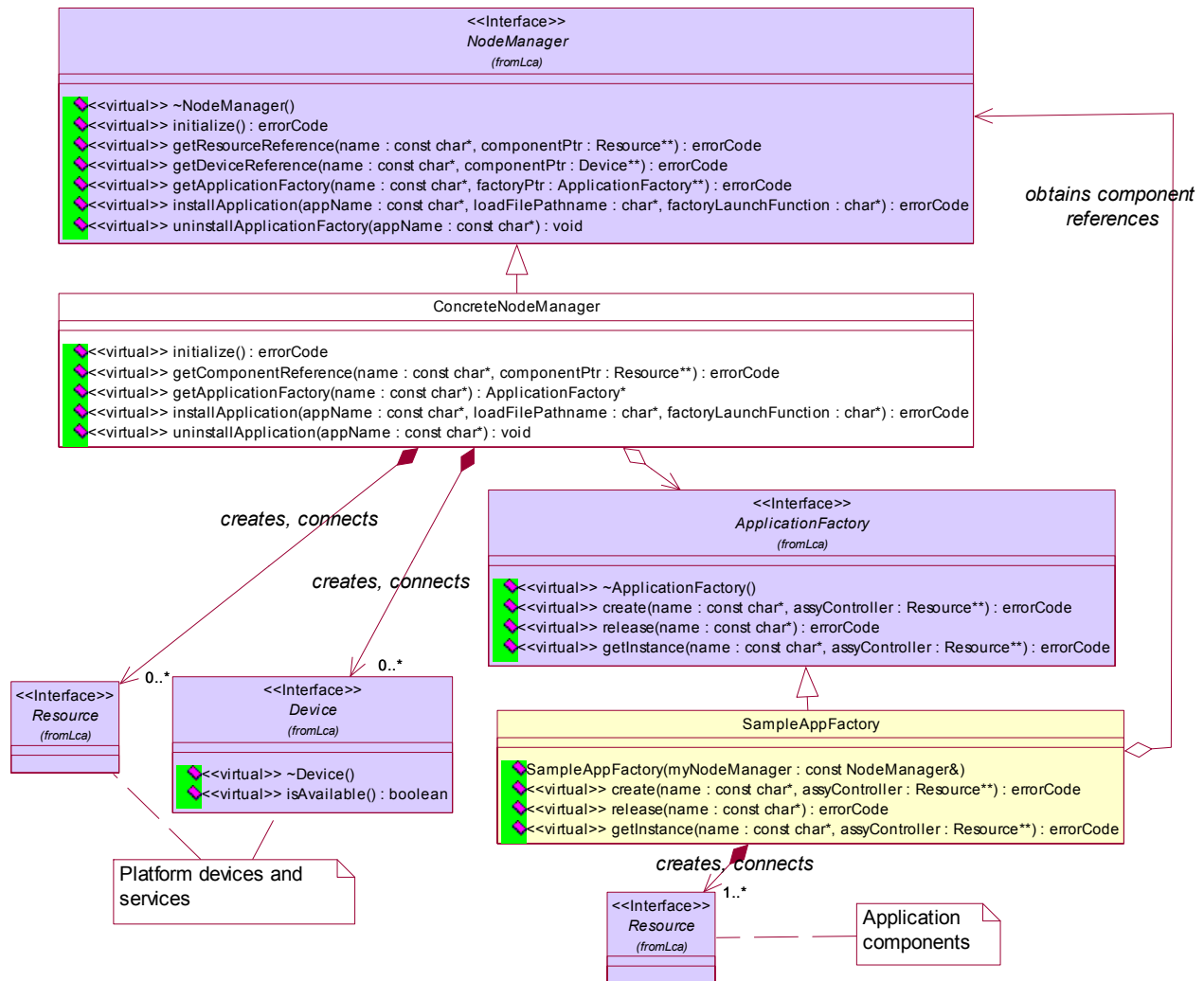


Figure 2: LCA Management Interfaces

4.1 Pre-installed or dynamically installed applications

While the *NodeManager* is responsible for ensuring that an *ApplicationFactory* instance be created for each application, installation of individual applications is an optional capability. In a minimal LCA, if all applications are known in advance, a *NodeManager* implementation may be used that is essentially hard-coded to create the required *ApplicationFactory* instances. This is not only simple, but allows application code to be pre-linked into the platform executable for fast loading.

Alternatively, the *NodeManager* can be implemented to support application installation. In this scenario, all application code is linked into a shared library, including a named function to create an *ApplicationFactory* instance. When this shared library is installed, the *NodeManager* is passed the pathname to the shared library file as well as the function name to call to create the *ApplicationFactory*.

Upon boot, the *NodeManager* iterates through all such installed files, calling the named function, which creates the *ApplicationFactory* and returns a pointer to the created instance.

4.2 Static or Dynamic application instantiation

The LCA intentionally does not specify behavior for the *ApplicationFactory::create()* and *release()* operations – only the postconditions. Since CORBA is not used, application memory footprint is often quite small and hence the value of releasing components may be scant. Because of this, an *ApplicationFactory* implementation could choose to create the application at *ApplicationFactory* creation (or first application create), leave it permanently created, and only perform port connect and disconnect operations when the *create()* and *release()* operations are called. This can result in almost instantaneous application launch times.

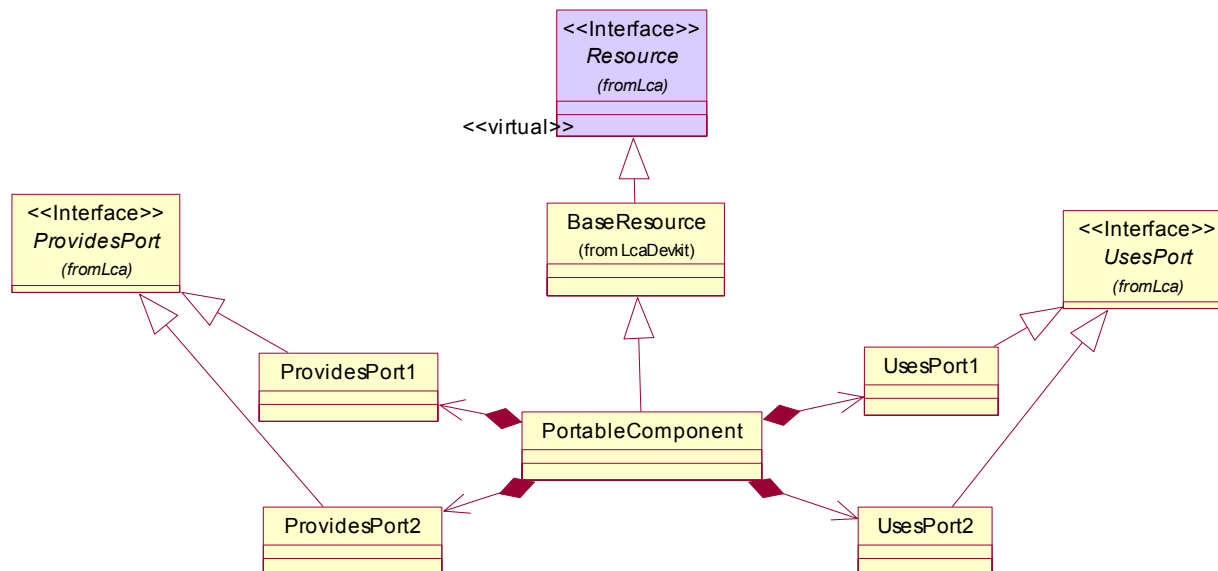


Figure 3: Component structure for SCA portability

Alternatively, the SCA-style method of dynamic creation and destruction can be used.

4.3 Hard-coded vs. scripted creation

While the *NodeManager* is responsible for creation of *Devices* and platform *Resources* (“services”), this creation can similarly be hard-coded into the *NodeManager* implementation, or scripted via XML or other means, and interpreted by the *NodeManager* implementation. A similar option is seen with the *ApplicationFactory* – either a generic component running off scripts or hard-coded specialized classes may be used in implementation, allowing a trade between small core framework size vs. versatility.

5. APPLICATION PORTABILITY TO/FROM SCA

Facilitating cross-portability of applications between LCA and SCA environments was a major goal for the LCA design, and is primarily accomplished by using a similar component model. To further aid this portability a combination of “development kit” base classes and a composite port structure can be employed, as shown in Figure 3. In this pattern, common in the JTRS community and somewhat simplified for clarity, the actual inter-component communication protocols are concentrated in standalone port classes. A core component then holds these port classes by composition. The port classes themselves are specialized to LCA or SCA, and in the case of SCA contain most of the CORBA dependencies. Communication between the core component and port classes is done exclusively using C++. In this way, the base component

can be ported between LCA and SCA environments by (mostly) swapping port classes. A similar separation is done with the framework state model by wrapping standard behavior in a *BaseResource* helper class.

While this approach greatly facilitates porting, residual effort remains, as CORBA employs pass-by-value semantics using CORBA-specific data structures (e.g. sequence containers), while LCA permits the standard embedded “pass by pointer, use or copy data before return” semantics. While it would be possible to write the core component using a completely neutral technology (e.g. using STL structures), SCA compliance issues and translation efficiencies preclude such use in most situations.

Differences between CORBA and LCA threading and synchronization models also need examination. While CORBA provides multiple synchronization models, LCA communication is always done at the “local C++ call” level. From a component synchronization model, this most closely matches a CORBA two-way call, which luckily is the most common in SCA systems, and always present in process co-located CORBA components. If true CORBA one-way semantics are required, a *ProvidesPort* class with an internal thread and queue could be employed to provide similar synchronization.

In summary, application portability between LCA and SCA, while not all-embracing, can typically be achieved with less than a 5 percent code modification level (for the core component classes) with careful design. Examination of the JPEO device APIs [2] also show little difficulty, as they typically employ two-way calls, relatively little container-based data passing, and few if any CORBA any types.

6. LCA PERFORMANCE

Since the LCA specification, like the SCA, consists of a set of interfaces and conventions instead of an executable code, LCA sizing and performance is very implementation specific. Based on our experience in employing the LCA in our Harris UNITY™ XG-100 Public Safety radio, overhead and code was found to be truly minimal. Employing a C++ hard-coded (vs. XML parsing) implementation, the XG-100 *NodeManager* creates / initializes / starts four *ApplicationFactories*, 7 *Devices*, and 8 *Services* (platform *Resources*) and makes 52 port connections. This *NodeManager* consumes 18kB of code (.text) memory, creating all platform components and factories in approximately 360 milliseconds². Most of this time is actually spent in the components themselves, not by *NodeManager* execution, which is presumed to be a small fraction thereof.

From an application standpoint, the typical LCA .text size overhead for a *Resource* (using a LCA *BaseResource* DevKit class) is less than 1 kB, exclusive of component-specific code logic. A typical *ApplicationFactory* (in this case launching and initializing 1 component and getting / connecting and disconnecting 9 ports) consumes 5 kB of .text. The execution of this factory (exclusive of actual waveform initialization) is trivial in the millisecond range. Waveform teardown is similarly trivial, as in this implementation the application is left in memory, and only stopped and disconnected.

In total, the XG-100 radio performs a full cold-boot from power off to P25 receive audio in less than 3 seconds, a significant savings compared to full-SCA based solutions.

7. OTHER APPLICABLE FRAMEWORKS

Software-based radio technology is not particularly new, although formalized frameworks are a more recent phenomenon. Beyond the SCA itself, the Object Management Group (OMG) Swradio specification [3] is the most comprehensive, using a Model-Driven-Architecture (MDA) approach to define a SCA compatible framework, with (potentially) multiple Platform-specific Metamodel (PSMs) to map to different implementation technologies. While broad and arguable more scalable than the SCA, most applications are still best suited for the same platform range as the SCA.

Lafaye and Nicolle[4] define a DSP-environment extension of the Swradio framework that shares many essential characteristics with LCA. In this framework non-CORBA components supporting SCA-like operations

(Cpp_ResourceComponents) are hosted in a micro-framework (DSP μ F), which provides connection services. This framework, however is defined only within the larger context of a CORBA-based Swradio implementation.

Much additional work has been done on DSP and FPGA implementations of subsets of the SCA, primarily for use within an SCA system to enhance component portability. These include DSP-optimized CORBA implementations [5] and FPGA component structures that can interconnect with SCA frameworks.

8. CONCLUSIONS

While the SCA continues to be the most appropriate radio framework for larger platforms, the LCA is tailored with a “sweet spot” towards simpler, single processor systems. Based on Harris’ experience, it has accomplished its primary goals, which is to provide a standard, versatile component framework that not only is familiar to SCA developers, but also which can provide a measure of application portability between the two environments. Furthermore, flexibility in implementation will support both customized as well as reusable script-driven core frameworks.

9. REFERENCES

- [1] Joint Program Executive Office JTRS, “Software Communications Architecture Specification v2.2.2”, <http://sca.jpeojtrs.mil/>, 15 May, 2006.
- [2] Joint Program Office Executive Office JTRS, “APIs Release 1.1.1”, <http://sca.jpeojtrs.mil>
- [3] Object Management Group (OMG), “PIM and PSM for Software Components Specification (formal/07-03-01)”, <http://omg.org>, 1 March, 2007.
- [4] F. Lafaye, E. Niccolet, “A Dsp Micro-Framework (DSP μ F) for OMG Swradio Specification Extension”, SDR Forum Technical Conference 2007 Proceedings, November, 2007.
- [5] J. Bickle, “Next Generation SCA Operating Environments”, SDR Forum Technical Conference 2006 Proceedings, November, 2006.

² The processor type and clock speed is not publically releasable, but is consistent with higher-end low-SWAP public safety radios.