TAKING THE SCA TO NEW FRONTIERS

Steve Bernier (CRC, Ottawa, Ontario, Canada; steve.bernier@crc.ca)
Claude Bélisle (CRC, Ottawa, Ontario, Canada; mailto:claude.belisle@crc.ca)
Communications Research Centre Canada (CRC)
3701 Carling Ave., PO Box 11490, Station H
Ottawa ON K2H 8S2
Government of Canada

ABSTRACT

The Software Communications Architecture (SCA) has been developed by the US Department of Defence in the late 1990's to respond to an urgent requirement to standardize the development of their radio equipment. The SCA has now been adopted throughout the world by military organizations as the foundation for their radio development.

The SCA however is not, and should not be considered, a military specific architecture. The SCA, and the now available associated development tools, truly form a component-based development architecture, so popular for Business-to-Business applications.

There are however still some reluctance in using the SCA outside the military market. In this paper, we first explain how the SCA can be seen as a CBD framework and how it differs from other popular CBD frameworks. We will then cover the Myths and Realities of the SCA and demonstrate that the SCA is well suite for not only public safety and commercial radio systems but has applicability in almost any embedded systems, from space to avionics, automobile, radar, test equipment and other electronic devices.

1. INTRODUCTION

As defined by the creators of the specification, the SCA is more or less a framework that standardizes the development of signal processing platforms and applications to simplify their integration. The original goal was to provide the US DoD with radio sets for which:

- applications could be easily ported from one platform to another to enhance communications interoperability;
- commercial-off-the-shelf (COTS) technology could be easily integrated, to reduce development and maintenance cost;
- the relation between the hardware platforms and the software applications could be abstracted, to simplify the integration and testing phases.

Rather than creating yet another framework from a white sheet, the SCA has been built by assembling commercially available software standards:

- POSIX (Portable Operating System Interfaces) offers code portability
- CORBA abstracts inter-process communications
- CCM (CORBA Component Model) provides a development life cycle structure
- X.731 ITU/CCITT OSI provides device state Management

The concept behind the SCA is one of development by components, software and hardware, with a key emphasis on a set of rules and behaviour to facilitate the integration of these components together.

This approach is the foundation of Component-Based Development (CBD).

2. CBD FOR EMBEDDED SYSTEMS

CBD is a programming trend that started some ten years ago and has reached an unprecedented level over the past three years, with the most popular CBD environment being Microsoft .Net and Sun Microsystems Enterprise Java Beans (EJB). However, unlike .Net and EJB, which require specific operating infrastructures (.Net mandates the use of Microsoft operating systems –Windows or Vista - and EJB requires a Java virtual machine) the SCA was designed to be a framework suitable for heterogeneous systems. The SCA is a platform-independent framework, supporting multiple operating systems, in their native form; multiple processor families; and a wide range of external devices.

While the SCA was developed to address the specific needs of the US DoD communications infrastructure, it has all the characteristics to be an architecture of choice for many other embedded system applications, from space, to avionics, automobile, medical, personal devices and other electronics systems.

The SCA standardizes two main categories of components, those forming an application, and those forming the hardware platform. It also defines a set APIs to support

deployment related functionalities and for making components "composable" which is at the heart of CBD. It does not specify any domain specific API, such as one for an RF synthesizer or an antenna positioning gimbal.

It is these domain specific APIs that define the domain and the type of applications that can be deployed as shown in the Figure 1.

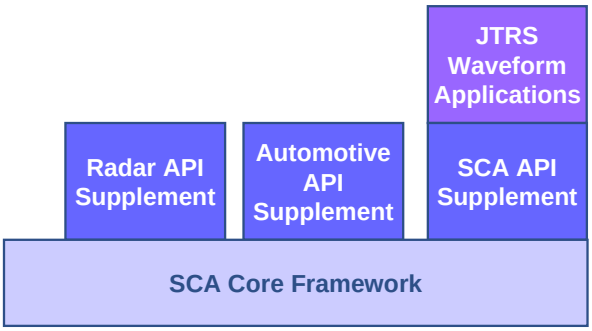

Figure 1. SCA Vertical Markets

One of the early issues with the SCA for embedded system was the requirement to use proprietary code to handle communications with specialized hardware devices (DSP, FPGA). HAL-C was produced but quickly became superfluous. With CORBA now available for DSP and FPGA, we can communicate directly to these devices and use them in similar fashion as for GPP.

The software communications architecture (SCA) has been developed by the US Department of Defense in the late 1990's to respond to an urgent requirement to standardize the development of their radio equipment. As created, the SCA is more or less a collection of commercially available software, assembled to offer an operating environment to radio developers.

The SCA has now been adopted throughout the world by military organizations and is also being used by many commercial organizations as the foundation for their radio development. A number of companies are now offering signal processing platforms, enabled with the SCA.

The SCA however is not, and should not be considered, a military specific architecture. The SCA, and the now available associated development tools, truly form a CBD architecture, so popular in the Business-to-Business community. However, by opposition to the Microsoft .Net, Sun EJB, or OMG CCM, the SCA is device centric and is targeted at applications with heterogeneous platforms.

As much as the SCA was a paradigm shift in the development of military radio, as much as the SCA can become a paradigm shift in the CBD world.

3. SCA MYTHS AND REALITIES

As discussed in the previous section, the SCA has all the attributes required for a development and deployment framework for any embedded system. It supports multiple operating systems and with the use of CORBA, it is truly agnostic of the hardware platform. However, there is still some reluctance to use it for non-military applications.

In this section, some of the most commonly heard complaints about the SCA and its usage will be discussed and shown to be simply myths.

3.1 Myth #1: The SCA is Slow to Boot

According to rumors, some SCA radios take up to up to 15 minutes to boot. It is agreed that software defined radios will, in most cases, take longer to become operational than hardware-only radios. The question here is: Is this due to the SCA or simply to the fact that software must be loaded? In order to get a better understanding of the boot time issue, this section describes the different steps required to complete the boot sequence.

a. First, upon power up, an SCA POSIX AEP [1] compliant operating system and its services (e.g. file system) must be started. This step requires copying the binary image of an OS kernel from permanent storage memory to the processor run-time memory and launching the kernel. Depending on the speed of the physical memory and the bus connecting it to the processor, this step can be very fast; especially with a real-time operating system (RTOS).

b. Once the operating system is started, the software components forming the SCA platform are launched. First, a CORBA naming service must be started. Second, a DomainManager and potentially several DeviceManagers are launched. Then, each DeviceManager launches a number of Device software components. As far as the OS is concerned, all that work amounts to launching a number of tasks. Here again, the speed of memory, bus, and processor can make a big difference. And of course, it also helps to use a very fast OS.

   During the SCA platform boot up, both the DomainManager and the DeviceManager will parse a number of XML files. Generally speaking, parsing an XML file can be slow and require a large amount of runtime memory. But there are many ways to implement those steps; some better than others. However, no matter how it's done, parsing XML files still requires file access and that is something a CF vendor cannot easily optimize. The choice of a file system type (say NFS) over another (say RAM FS) can have a huge impact on performance.

c. Last but not least, an application must be launched in order to bring the radio in operational mode. During that process, potentially several software components will be launched and many XML files will be parsed. In other words, the launch of an application is much the same as booting the node components. Therefore, it can benefit from the same solutions.

In summary, the speed at which an SCA radio can be booted is affected by a number of things. The hardware actually makes a difference. As explained earlier, the memory speed (storage, run-time, etc.) and buses to the processor can make the difference. Obviously, the speed of the processors also plays a major role. The speed of communications between SCA software components can also affect the boot time of a radio; but that is covered in the next section. Nevertheless, the boot time myth is slowly being put to rest as SCA radios are being deployed and are actually booting as fast as the legacy radios they are replacing [2] [3].

3.2 Myth #2: SCA Applications are Slow

One of the longer lasting myths about the SCA is that it is too slow because of CORBA. CORBA is the inter-process communications (IPC) mechanism which allows SCA components to interact and exchange information. Since SCA components are developed separately as black boxes, they must rely on an IPC mechanism to interact with each other. The SCA mandates the use of CORBA as the IPC. CORBA is actually a programming language and is processor independent. CORBA is also scalable as it provides a single model for communications between components whether they are located in the same process or across the network. In short, CORBA is great for portability, which is the main goal of the SCA after all.

However, CORBA has the very bad reputation of being slow. That reputation dates back to its early days when the General Inter-Orb Protocol (GIOP) was only implemented using TCP/IP (called Internet Inter-Orb Protocol – IIOP). GIOP is the protocol by which different CORBA objects interact. Fortunately, CORBA has come a long way since then.

ORBs can now be used for real-time embedded systems. COTS real-time ORBs provide a very fast implementation of IIOP; nearly as fast as if TCP/IP was used directly [4]. However, TCP/IP being inherently too slow for most real-time applications, it is also possible to use a different pluggable transport [5] which outperforms IIOP (see figure 2). In some cases, switching from the TCP/IP transport to a very fast transport can produce savings of one order of magnitude [6]. Real-time ORB vendors typically support a number of transports (UDP, multicast, shared memory). Some even support RTOS specific transports and embedded system interconnects like CompactPCI and VME [7].

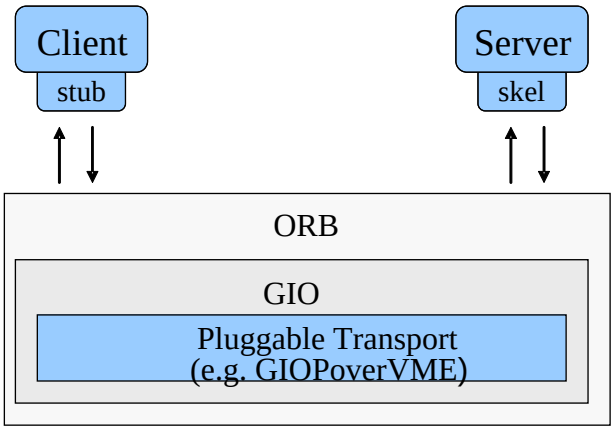

Figure 2 – CORBA Communications

It is a myth that the SCA is slow because of CORBA. As explained above, CORBA communications can be made as fast as the native platform transport [7]. Unfortunately, most CORBA developers are unaware of the concept of pluggable transports which contributes to keeping this myth alive.

3.3 Myth #3: The SCA is too fat

The SCA requires a POSIX AEP compliant operating system which theoretically takes more space than not using an operating system. The SCA also relies on CORBA which requires a translation layer (source code for stubs and skeletons) that, for the SCA, amounts to approximately 730kB of binary code using Objective Interface System ORBexpress or 3.3 MB using TAO (obtained using default configuration for respective IDL compilers). But more importantly, the SCA relies on XML files which typically require an XML parser. Such a parser is usually rather large in memory footprint. Using the open-source Xerces-C++ [8] XML parser to build an SCA core framework requires approximately 2.6 MB of static footprint and typically 4 MB of dynamic footprint.

Footprint is dependent of the number of components in the radio (platform and application). As an example, the ISR IDP-100 [9] runs simultaneously a Voice-over-IP and a streaming video application with approximately 51 MB of memory. That footprint also includes all the software of the operating environment, namely: the SCARI++ core framework (CRC), the platform SCA devices (ISR), ORBexpress and naming service (OIS), and INTEGRITY with file system and POSIX support (Green Hills Software).

Those memory requirements may seem large for small form factor platforms. CF implementers are improving and constantly making their product smaller and faster. However, it will always remain that implementing a waveform with software instead of hardware requires more memory. The memory requirements of a SDR can not be compared to the requirements of a radio which merely use software for implementing basic control functions. For the same reasons, the flexibility of a SDR cannot be compared to traditional radios. The cost of flexibility is memory; more than before but not a whole lot.

4. THE FUTURE OF THE SCA

The future core frameworks will be smaller and faster; that's no secret. But what's not so obvious is that there are two approaches for achieving those goals. Most of the research has been focused on optimizing the tasks to be executed for maximum speed or minimum footprint. Another approach consists in eliminating some tasks normally required for the deployment of components. This second technique will be referred to as static deployment. The remainder of this section describes both approaches.

4.1 Tasks Optimization

The tasks optimization approach is relatively straight forward. First, the sequence of task s performed during the deployment phase of the components is identified. Then each one of these tasks can be optimized. Optimizations need to maintain compliancy with the SCA requirements. A core framework typically has to perform several tasks to deploy any component so there is plenty to choose from for optimization.

Charles Linn [10] identified a number of different tasks that can be optimized for small platforms. Some optimizations actually require additional APIs to the SCA standard ones. For instance, speeding up file system access actually requires an extended API (to the SCA File interface) to get access to a native file name. However, eliminating the use of a DOM XML parser is an implementation level optimization which does not require API changes. Most tasks optimizations are relatively easy to implement and generally don't pose a certification problem.

4.2 Static Deployment Optimization

In the second approach, static deployment, the goal is to eliminate as many tasks as possible. A CF can achieve this by saving deployment context information and reusing it. Linn [10] describes one static deployment optimization. He explains how the resolved property values for components could be saved for future use. Each time a component is deployed, a core framework must determine the initial value to use for configuring each property. This is done using the SCA rules of precedence for property value overloading. If the resolved values were saved by the core framework, there would be no need to compute the second time a component is deployed. In other words, the property resolving task could be skipped.

Basically, any decision made by a core framework could be saved and used the next time it is required. For instance, when an application is used for the second time, a core framework could avoid redeploying it if the target devices used in the previous deployment support a caching feature. Avoiding the copy of component artifacts saves a significant amount of time especially when memory access is slow. Of course, there are cases where a core framework can't simply restart a previously deployed component. For instance, when a device's cache has been cleared or when a device doesn't support caching. In those cases, the core framework defaults back to deploying the components as usual.

Another example of a static deployment optimization is the transformation of indirect connections into direct connections. As explained in [11], a connection is indirect if at least one of the components involved in the connection (source or destination) is identified using run-time information. A direct connection is one where both components involved in the connection are identified by name or identifier. The current SCA supports three types of indirect identification mechanisms [12]: domainfinder, devicethatloadedthiscomponentref, and deviceusedbythiscomponentref. All three mechanisms require that a core framework gather deployment information which can then be used in lookup tables to identify a component. Core frameworks could save the result of the identification process and thus perform direct connections the next time the application is deployed.

Ultimately, a core framework could remember every decision it makes to deploy applications. This would allow applications to be redeployed skipping all the tasks except the actual instantiation of components, their configuration and their inter-connections. Static deployment doesn't require any special API from the components being deployed. SCA applications don't need to be modified to be deployed statically. Another important benefit of full static deployment is determinism. Redeploying an application is predictable. Those properties are very important for safety-critical embedded systems such as those used in aircrafts.

Static deployment optimizations may seem like they could lead to certification issues more easily than tasks optimizations. But a core framework typically does not skip any deployment tasks the first time an application is deployed. Thus, for a first deployment, there is no

difference in behavior between a new generation core framework and a legacy one. Consequently, there should be no certification problems.

As discussed, both optimizations approaches can provide significant improvements. Clearly, new generation core frameworks will provide a combination of static deployment and tasks optimizations. In fact, the latest version of the SCARI++ core framework [13] already provides some optimizations of both kinds.

5. CONCLUSION

The SCA is a component-based development environment with all the characteristics to be used well beyond military radio equipment. It is domain and platform agnostic design makes it a perfect candidate for any embedded system, from military and public safety radios to space, avionics, automobile and other commercial systems. Unlike other component-based development frameworks, it is not restricted to a specific operating environment. Since the SCA is mainly a framework for the deployment and configuration of applications, its impact on the signal processing performance is minimal. In fact one can say that it improves the performance as it allows the developer to choose the best processors and inter-process communications protocol.

Being an open specification, its evolution can be community driven, ensuring a rapid response to the market requirements. The SCA specification has lead to the creation of an ecosystem of SCA products and services. This allows SCA radio and application developers to be more productive since they can concentrate on their business logic. The SCA is slowly proving that it is a truly versatile component-based development environment for embedded systems.

6. REFERENCES

[1] JPEO/JTRS, SCA Application Environment Profile, SCA Specification Appendix B, April 2006.

[2] Press Release, *Thales JTRS Radio Achieves Government Certifications – First in Industry*, January 2006.

[3] Mark Turner, "*Harris SDR Solutions – Scalable, Reusable, and Secure*", International Software Radio, London, UK, June, 2004.

[4] C. Hrustich, "*CORBA for Real-Time, High Performance and Embedded Systems*", Fourth International Symposium on Object-Oriented Real-Time Distributed Computing, isorc, p. 345, 2001.

[5] D.C. Schmidt, C. O'Ryan, O. Othman, F. Kuhns, J. Parsons, "*Applying Patterns to Develop a Pluggable Protocols Framework for ORB Middleware*", Design Patterns in Communications, Cambridge University Press, 2001.

[6] J. Belzile, "*Putting it all together – Objectives and Challenges*", SDRF'05 Technical Conference, 2005.

[7] G. Middioni, "*CORBA over VMEbus Transport for Software Defined Radios*", www.motorola.com, 2005.

[8] Xerces C++ Parser, http://xml.apache.org/xerces-c/

[9] ISR IDP100 Development Kit, http://www.isr-t.com/products_idp.htm

[10] C. Linn, "*Designing Jtrs Core Frameworks For Battery-Powered Platforms: 10 Techniques For Success*", SDRF'04 Technical Conference, 2004

[11] F. Lévesque, C. Auger, S. Bernier, H. Latour, "*Jtrs Sca: Connecting Software Components*", SDRF'03 Technical Conference, 2003.

[12] JPEO/JTRS, Domain Profile, SCA Specification, Appendix D, April 2006.

[13] SCARI++, http://www.crc.ca/rars.