JTRS SCA: CONNECTING SOFTWARE COMPONENTS

François Lévesque (CRC, Ottawa, Ontario, Canada, francois.levesque@crc.ca)
Charles Auger (CRC, Ottawa, Ontario, Canada, charles.auger@crc.ca)
Steve Bernier (CRC, Ottawa, Ontario, Canada, steve.bernier@crc.ca)
Hugues Latour (CRC, Ottawa, Ontario, Canada, hugues.latour@crc.ca)

ABSTRACT

The portability of applications between different radio platforms is one important benefit of the JTRS Software Communications Architecture (SCA). Over the years, the SCA portability features have certainly been greatly improved. The deployment process for SCA applications is without a doubt the most important feature for portability. However, the specification of interconnections between the components of an application is also crucial and often overlooked.

The SCA provides a wide range of possibilities for establishing connections between software components but certain options are only valid for specific situations and, amongst the valid options, some may lead to portability problems. The SCA does not provide any guidance along those lines. Finally, the SCA only provides limited support for interconnecting applications.

In this paper, options for establishing connections between software components are explored and categorized according to their usage. The more portable options for interconnecting components of an application are identified and discussed. Also, a proposal to allow connections between applications, within the SCA framework, is presented. Finally, this paper concludes on a discussion regarding the need for the SCA to provide support for aggregated applications.

1. INTRODUCTION

When referring to computer software, portability is a term used to describe how easy it is to change a software application for being used with a different hardware platform and/or operating system. In the SCA world, portability refers to the capacity of an SCA application to be used on different SCA platforms. In general, software portability can be obtained using one of the following approaches:

- The source code of a portable component is sent to an interpreter program that behaves appropriately for the host platform. Each different platform must provide an interpreter.
- The source code of a portable component is compiled for a specific platform and is executed by a virtual machine that behaves appropriately for the host platform. Each different platform must provide a virtual machine.
- The source code of a portable component is compiled for each different host platforms and is executed natively. This process requires a capability exchange mechanism in order to dynamically choose the proper binary code based on the capabilities of a specific host platform.

The SCA uses the multiple-compiles model to achieve portability. SCA components (e.g. *Devices* and *Resources*) are compiled for the different platforms in which they are intended to be used. The SCA specification requires software components to provide a description of their requirements and capabilities, which are compared during the process of software deployment. The requirements and capabilities are described using XML and formatted as specified in the SCA Domain Profile document [1].

Although this portability model allows the appropriate implementation of an SCA component to be mapped to an SCA platform, it doesn't deal with the portability of the connections that must be established between components of an application or between devices/services of a radio node. If it is not used properly, the SCA connection model can lead to portability problems. In the following sections, the SCA connection model is presented, and guidelines aimed at achieving portability are presented.

2. SCA CONNECTION MODEL

In the SCA specification, connections are used to provide references to components. This concept allows a component to obtain a reference to another component for communication and control purposes. For example, in a connection where component_A is the source and component_B is the destination, component_A obtains a reference to component_B and may invoke some method to provide data to be processed by component_B. However, since connections are unidirectional in the SCA, component_B is not given a reference to component_A unless a second connection is defined for this purpose. For the source component to receive a reference to the destination component of a connection, it must provide a special Applications Programming Interface (API) named *Port*. To establish a connection, the *connectPort* method is invoked on the source component with a reference to the destination component as an input parameter.

Note that the orientation of an SCA connection does not indicate the direction of data flow between two components. In the previous example, component_A is given a reference to component_B which could suggest that data will flow from component_A to component_B. However, component_A can choose to request data from component_B in which case the direction of the data flow is different from the connection orientation. Finally, even if connections are unidirectional, the data flow between two components can actually be bidirectional; component_A can send/request data to/from component_B.

In the SCA there are two types of software assemblies that may require component interconnections: node and application. A SCA radio can be composed of many nodes, each of which must provide a node assembly descriptor. The node assembly descriptor (called DCD) indicates which components must be started in a radio node and how they must be connected to each other. For example, an analog-to-digital *Device* may need to be connected to a timer *Device*. Another example would be a connection between an *ExecutableDevice* and a *Log* service of a same radio node. An SCA application is composed of many components, each of which performs a certain number of signal processing functions. These components must be connected to each other in order to perform the aggregated behavior of an application. An application is described using an assembly descriptor (called SAD) that indicates which components must be launched and how they must be connected.

In SCA version 2.2, the components of a node assembly are launched by a *DeviceManager* whereas the connections between components are established by a different entity named *DomainManager*. In the case of an application, the *ApplicationFactory* is responsible for both launching and connecting the components of the assembly. This distinction is key to understand the different type of connections that may be established for a node and for an application.

The SCA defines two types of connection; port-to-component and port-to-port. As mentioned earlier, the source of a connection is always a component that implements the *Port* interface. However, the destination of a connection may be the destination component itself (port-to-component) or a sub-component obtained via a port of the destination component (port-to-port). The difference between the two types of destination is subtle and is related to how the destination component provides the interface needed in the connection. As explained previously, the destination of a connection is a component with an interface that will be used by the source component to either receive or provide data. If the destination component implements the needed interface by aggregation, connections will have to be established to the sub-component implementing the interface. However, if the destination component inherits the needed interface, connections will be established directly to the component.

The following pseudo-code shows the steps required to establish a port-to-component connection between two components.

```
portA1 = component_A.getPort("Port1")
portA1.connectPort(component_B, "toB")
```

The following pseudo-code shows the steps required to establish a port-to-port connection between two components.

```
portA1 = component_A.getPort("Port1")
portB1 = component_B.getPort("Port1")
portA1.connectPort(portB1, "toB1")
```

In these two examples, it is assumed that references to component_A and component_B were previously obtained. The SCA offers five mechanisms for obtaining/identifying the source or destination component of a connection. They can be categorized into "direct" and "indirect" identification mechanisms as described below.

2.1. Direct Identification Mechanisms

Direct identification mechanisms allow source or destination components to be identified using pre-defined and static information. The direct identification mechanisms are:

- **Naming Service Name**: Although it is only mandatory for *Resources*, any SCA component can register to the CORBA naming service. The name used by a *Resource* for registration is composed of the name and id attributes found in the application assembly descriptor (SAD *findcomponent.namingservice.name and SAD partitioning.componentinstantiation.id*). The *Resource* name can be used in a connection definition (*findby.namingservice*) to identify the source or destination of a connection. Depending on the type of software assembly, either the *DomainManager* or the *ApplicationFactory* will query the CORBA naming service to obtain an object reference to a component.

- **Component Instantiation Reference**: In a software assembly descriptor, each component instance is associated to a unique identifier (*componentinstantiation.id*). This identifier can be used to identify the source or destination component of a connection (*componentinstantiationref.refid*). The *ApplicationFactory* remembers the component identifiers as there are launched and uses this information to establish connections between components of an application.

2.2. Indirect Identification Mechanisms

Indirect identification mechanisms offer an alternative to using a component's unique identifier or name to identify source or destination components of a connection. While direct identification mechanisms refer to static information known at development time, indirect identification mechanisms provide access to runtime information. The indirect identification mechanisms are:

- **Domain Finder**: The domain finder identification mechanism can be used to establish connections to radio services (e.g. log, naming service). Services from all nodes register to the *DomainManager* using a name and a type. Since the component that implements a radio service may be different in each radio, the component cannot be directly identified. Therefore, connections to radio services should be performed using a service type and, optionally, a name. Connections to services are specified in the assembly descriptors using the *findby.domainfinder* element.

- **Device That Loaded a Component**: This mechanism allows a connection with a *Device* that was used to load a specific component. For example, a component may request a connection to the FPGA *Device* that was used to load a specific algorithm. Since the *Device* used to load a component may change every time the application is deployed (due to capacity state), the connection to such a *Device* is made using the identifier of the component being loaded (*devicethatloadedthiscomponentref.refid*).

- **Device Used by a Component**: This identification mechanism allows a connection with a *Device* that is being used by a specific component. For example, an application Component_A may request to be connected to audio *Device* being used by the application Component_B. For this to happen, Component_B must first declare its need for an audio *Device* and associate it with an identifier to which Component_A will refer for being connected. Component_B declares its need through requirements in terms of capabilities and capacities. These requirements will be satisfied during the launch of Component_B and may change from time to time. Therefore, Component_A may sometime be connected to a different audio *Device*, but it will always be connected to the audio *Device* used by Component_B.

3. SCA CONNECTIONS RESTRICTIONS

As mentioned in section 2, connections are either performed by an *ApplicationFactory* or by the *DomainManager*. An *ApplicationFactory* is responsible for establishing connections between components of an application or between an application component and a *Device*. Since the *ApplicationFactory* is also responsible for launching the components, it can gather sufficient information for allowing the use of all five mechanisms for the identification of the source and the destination components of a connection.

However, because the *DeviceManager* is responsible for launching the components of a node, the *DomainManager* cannot obtain crucial information for certain identification mechanisms. In the current version of the SCA (2.2), the information gathered by the *DeviceManager* cannot be provided to the *DomainManager* because of the lack of an API. Furthermore, since a radio manufacturer may purchase the *DeviceManager* and the *DomainManager* from different vendors, it is not practical to create non-standard extended APIs to address this issue. Therefore, the connections specified between node components cannot use the following identification mecanishms: *devicethatloadedthiscomponentref* and *deviceusedbythiscomponentref*.

4. CONNECTION PORTABILITY ISSUES

Application portability is an important feature of the SCA. However, providing many implementations of an application is not sufficient to achieve portability. For an application to function properly, all the connections between the components must be established. Therefore, the connections required for an application must be portable. Issues that could preclude connection establishment on some platforms supported by an application are: direct references to *Devices*, different Radio Frequency (RF) chain implementations, connections between applications, components port names, and references to a *Device* used by an implementation of a component.

4.1. References to Devices

Typically, an application has some components that need to use *Devices* for data input/output purposes. Since the name of a Device is chosen by a radio integrator, it may differ in each radio. Therefore, using a direct identification mechanism is not a portable solution unless *Device* names are standardized. However, it is much easier to standardize on a common subset of *Device* characteristics (capabilities and capacities) [2]. Therefore, a better approach for establishing a connection with such a *Device* is to have the component, which needs access to it, declare a "use" association. The association with a *Device* is specified based on requirements. This "use" relationship would be satisfied upon launch time and could then be used by the same component for requesting a connection with the *Device* using the *deviceusedbythiscomponentref* indirect identification mechanism.

4.2. Different RF chain implementations

An application that acquires its data packets from a RF must have access to some RF *Devices*. Typically, the RF signal arriving from the antenna must be converted to digital data through an analog-to-digital (A/D) *Device*. If the RF is too high relative to the A/D sampling rate, it needs to be down-converted to an intermediate frequency (IF) prior to the conversion to digital data. Finally, in many cases the sample data rate coming from the A/D may have to be lowered for the processing of the baseband data. This is typically done through digital down-conversion/decimation. The steps that need to be performed for the conversion of RF signals to baseband data sample can be implemented in various ways which means different radios could provide different groups of RF *Devices* and still offer an equivalent service.

Figure 1 shows a scenario where the RF from the antenna is converted into digital data through an A/D, and then down-converted/decimated to baseband data through a Digital Down Converter (DDC). In this scenario, the A/D and the DDC are configurable SCA *Devices*. Figure 2 shows a different RF chain where the RF is down-converted to IF by a configurable analog hardware and then converted to baseband data by an A/D. In this case, the analog down-converter, the A/D, and the DDC are configurable SCA *Devices*.

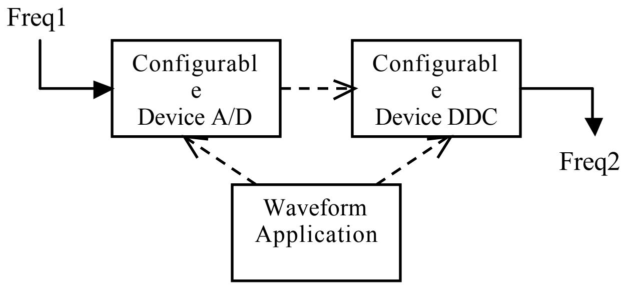

Figure 1

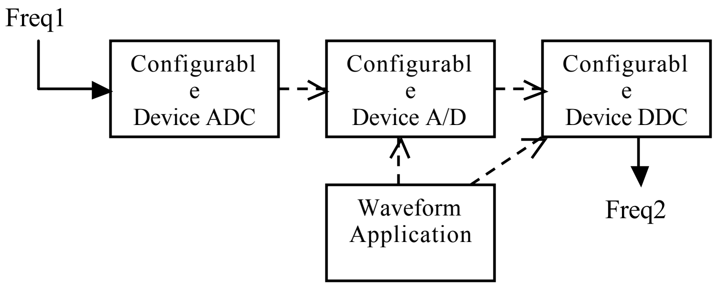

Figure 2

In both scenarios, the waveform application would have to configure the SCA *Devices* to set their receiving and transmitting frequencies, data rate, etc. However, in the first scenario, there are two *Devices* to configure while in the second, there are three. To configure a *Device*, the waveform application needs to be connected to it in order to obtain its object reference. Since the number of *Devices* varies in the two scenarios, it is impossible to define a portable application assembly descriptor because of the different requirements with regards to connections.

To allow portability of the connections in such scenarios, an abstraction of the RF hardware *Devices* could be used. A high-level abstraction RF *Device* component could hide the details of the radio RF chain. Application components would not connect to individual RF Devices but only to the high-level abstraction RF *Device*. Such an abstraction is being defined by the Object Management Group [3] while API definitions for all kinds of *Devices* are being defined by the Systems Interface Working Group of the SDR Forum [4].

4.3. Connections between applications

Examples of SCA applications include waveforms for radio communications, network layer applications over which other applications can operate, or cross-banding applications. Some of these examples require that applications communicate with each other in order to provide a single aggregated functionality to the radio user.

The current version of the SCA (2.2) doesn't specify how applications may be connected to each other. It is however possible to connect two components of different applications using their name as registered in the CORBA naming service. The main problem with this approach is that the name of an application component is always concatenated to the name of the application, which is typically chosen at run time by the radio operator. Therefore, this kind of connection requires that the operator use predefined names for the interconnected applications. In addition, connecting to components of an application actually breaks the concept of application encapsulation. SCA applications should not expose their internal structure for reasons of security and portability.

4.4. Components port names

To connect to *Devices*, applications must use port names which are defined by the *Device* developer. It is therefore possible that port names change from one radio to another. For this reason, port names for SCA components selected during application deployment time should be standardized to preserve the portability of the connections.

4.5. Association between a component and a *Device*

As mentioned above, a connection can be specified with a *deviceusedbythiscomponentref* to indirectly identify a *Device* being used by a specific component. In the SCA, the declaration of a "use" relationship can be defined globally for a component (component level). It can also be defined for a specific platform supported by the component (implementation level). However, if a connection references a "use" relationship not defined for all implementations, the connection establishment could fail depending on the chosen implementation. Consequently, if a "use" relationship is defined at the implementation level, it should be defined for all implementations.

5. MAXIMIZING CONNECTION PORTABILITY

In order to increase the portability of application connections, some rules should be followed for the definition of connections:

- A *Device* involved in an application connection should be referenced by using the *devicethatloadedthiscomponentref* or *deviceusedbythiscomponentref* XML elements in order to allow the use of the indirect identification mechanisms.

- Every *Device* used by an application should be defined as a "use" relationship for the assembly controller component which contains the application specific behavior. Every connection to a *Device* should be done using the *deviceusedbythiscomponentref* identification mechanism for better portability. The fact that the assembly controller contains the "use" relationships increases portability because this is the only component that must be changed for every new application. Therefore, if an application component is replaced by another one, the "use" relationship upon which connections may depend will not be lost.

- If a *Device* used by a component is involved in a connection, the "use" relationship should be defined at the component level rather than at the implementation level.

- No connection should be allowed from external applications (e.g. user interface) to the components of an application. External applications should connect to the external ports of an application. The external ports, defined in the application assembly descriptor, should be mapped to the ports of the assembly controller to allow connections without breaking the encapsulation. This measure isolates an application user from the application's internal structure which can consequently be changed without affecting portability.

6. INTER-APPLICATION CONNECTION

As mentioned in section 4.3, some applications need to communicate together in order to provide an aggregated functionality to the radio user. An application can potentially be connected to another application since it implements the *PortSupplier* interface. However, there are two problems preventing portable connections between applications.

First, each application is created by an *ApplicationFactory* which can only create one type of application. Therefore, when an application needs to be connected to another application, the second application must already have been created. Since the order in which applications are created is controlled by the radio operator, it is impossible to automate the launch of two (or more) applications in a specific sequence. The second problem, as mentioned in section 4.3, comes from the fact that the encapsulation of an application must be broken to allow a connection between two applications. This approach limits the portability of connections since the structure of an application may change over time.

Under these constraints, the only way to connect two applications within an SCA radio is that each application be implemented as a single *Resource* and launched together from the same SAD. But this would require that an SCA application be created for every possible combination (i.e. application-to-application connection), which is simply impractical. Moreover, this approach prevents an application designer from defining fine-grained reusable components.

6.1. Potential Solutions

A first solution could consist in having one of the applications involved in a connection register as a radio service (e.g. network layer application). With this approach, the connection could identify this application using the *domainfinder* identification mechanism. This would however require a new type of service called "application" to indicate that the connection requires a reference to an application. This solution would preserve application encapsulation which is good for maintenance and portability. The drawback is that the application registering as a service would have to be created before the second application involved in the connection. In addition, the name of the second application would have to always be the same.

A better solution would be to add support for the concept of aggregate applications to the SCA. An aggregate application is composed of two or more applications. Just as an application is composed of several components, each performing a specific task in the data processing of the application, an aggregate application is composed of several applications, each implementing a self-contained task. For example, an aggregate application could be a bridge between two different waveforms.

To include this concept in the SCA, the *ApplicationFactory* behavior could be altered to launch an aggregate application in addition to the standard applications. A new assembly descriptor would be needed to indicate which applications compose the aggregate application and how they must be inter-connected. With this solution, the radio operator would only select the name of the aggregate application which would automatically be used for assigning names to the aggregated applications. In addition, connections would be established between applications, thus preserving encapsulation. Finally, this solution is relatively simple since it is a generalization of the existing SCA concepts.

7. SUMMARY

One of the goals of the JTRS Software Communications Architecture is to achieve waveform application portability. The specification of interconnections between components of an SCA application is an aspect that may influence the portability of the application, and should therefore not be overlooked.

In the SCA, the concept of connection is used to provide references to components for communication purposes. SCA v2.2 offers five ways of identifying the source or destination component of a connection, some of which may or may not be used depending if the connection must be established for a node or an application.

This paper presented and analyzed the SCA connection model in regards to connection portability. Portability issues that can preclude connection establishment have been described and recommendations have been made to maximize the portability of connections. Finally, the concept of aggregate application was introduced as a solution to the lack of inter-application connection mechanisms.

8. REFERENCES

[1] Software Communications Architecture Specification, Appendix D - Domain Profile (MSRC-5000SCA Appendix D rev 2.2)

[2] Software Communications Architecture Specification, Attachment 2 To Appendix D – Common Properties Definitions (MSRC-5000SCA Attachment 2 to Appendix D rev 2.2)

[3] Object Management Group, Software Radio Domain Special Interest Group, http://swradio.omg.org/swradio_info.htm

[4] Software Defined Radio Forum, System Interface Working Group, http://www.sdrforum.org/tech_comm.html