CROSS - A DISTRIBUTED AND MODULAR COGNITIVE RADIO FRAMEWORK

Benjamin Hilburn (MPRG: VT, Blacksburg, VA, USA; bhilburn@vt.edu); Warren Rodgers (Georgia Tech, Atlanta, GA, USA; wrodgers3@gatech.edu); Timothy R. Newman (MPRG: VT, Blacksburg, VA, USA; trnewman@vt.edu) and Tamal Bose (MPRG: VT, Blacksburg, VA, USA; tbose@vt.edu)

ABSTRACT

In cognitive radio systems with multiple components, the radio must possess a way to manage, control, and properly use the services provided to it by these components. This scenario grows more complex as components of the same type (e.g. multiple cognitive engines) are attached to a single radio, forcing the radio to make requests to the proper component. Additionally, in a cognitive radio with a non-static mission, the radio must be capable of adapting its use of components to achieve its mission goals. In order to make such a distributed system work, there must be a well-designed and strict API for both interfacing components and controlling the radio itself. Ergo, a distributed cognitive radio API must define both the networking protocol with which the components talk to each other, and function hooks that the client can use to control and interface with the radio. In distributed systems, this API must not require a static radio waveform, as the types and numbers of components present might vary from one radio implementation to another. As part of the Virginia Tech Cognitive Radio Open Source Systems project (VTCROSS) currently underway at Virginia Tech, we have designed a cognitive radio API that defines interfaces for all radio components and the component inter-communication protocol. We have also designed a Service Management Layer (SML) for managing multiple components, possibly distributed, for any single cognitive radio system. In this paper, we will present our design of component APIs and a SML for cognitive radio systems. We will also discuss how such a design can be used to create implementation-independent cognitive radios for any platform and mission.

1. INTRODUCTION

Software-Defined Radio (SDR) has become a promising solution to many of the primary issues confronting radio researchers today. One of the most heavily studied of these issues is the finite spectrum dilemma, which is certain to stagnate both wireless technology growth and adoption should it go unresolved. A potential solution to this problem is cognitive radio, which has been the focus of much research and discussion in recent years.

As a result of the large amount of attention SDR has received, a number of SDR frameworks, toolsets, and platforms have emerged – some open (e.g. [1][2]), some proprietary (e.g. [3][4]). Cognitive radio implementations are then built on top of these SDR frameworks, and are generally specific to that particular SDR platform and radio system. Ergo, collaboration and comparative research for cognitive radios is made difficult by a lack of interoperability and design compatibility.

To facilitate cognitive radio research, development, and prototyping, the Mobile & Portable Radio Research Group at Virginia Tech has created the Virginia Tech Cognitive Radio Open Source Systems project (VTCROSS) [5]. The goal of VTCROSS is to provide a distributed and modular cognitive radio system framework that sits atop an SDR platform. A component developed for one VTCROSS system will work with any other, regardless of what SDR or hardware platform it is deployed to.

Two major facets of this design are the interface between VTCROSS components and the Service Management Layer, which provides a service-oriented architecture for VTCROSS radio systems. In this paper, we present the VTCROSS architecture, the API we have designed for VTCROSS component development, and the communication interfaces that the components use during system operation.

2. NAMING CONVENTIONS

Due to the distributed nature of VTCROSS radios, we use specific naming conventions to distinguish between different aspects of the radio.

CROSS, on its own, is a framework. Using this framework, a client can build a complex radio. However, by itself, CROSS does not comprise a radio.

Since CROSS radios are generally distributed, simply referring to it as 'the radio' can be misleading. It is not clear whether the reference is to the radio hardware, the entire CROSS system, just a single CROSS component, or the host platform. For that reason, we refer to the working radio as a CROSS radio system' - the 'system' keyword denoting that the radio itself is comprised of many components, some complex and some simple. By default,

a CROSS radio system is not necessarily cognitive. It only becomes a cognitive radio once a cognitive engine component is connected to the system. Once this occurs, we call the entire radio a CROSS cognitive radio system'.

We refer to the radio design that CROSS defines as the 'radio architecture'. A suitable illustration of the CROSS architecture can be seen in Figure 1 - it is the block diagram that describes how a CROSS system is connected.

The CROSS project itself is free and open source. Hence, development can occur in any group, and forks can be created by anyone. Hence, VTCROSS and CROSS are sometimes interchanged, but we use the 'VTCROSS' to distinguish between outside groups and the core founding group of developers located at Virginia Tech.

3. CROSS ARCHITECTURE

The CROSS project implements a modular cognitive radio framework that provides portability and interoperability between components that may be independently developed for different platforms with different programming languages. This allows for flexible development of the cognitive radio system and allows developers to focus entirely on their radio component of choice without the need to spend time developing or modifying components that have no relevance to their specific focus of research or testing.

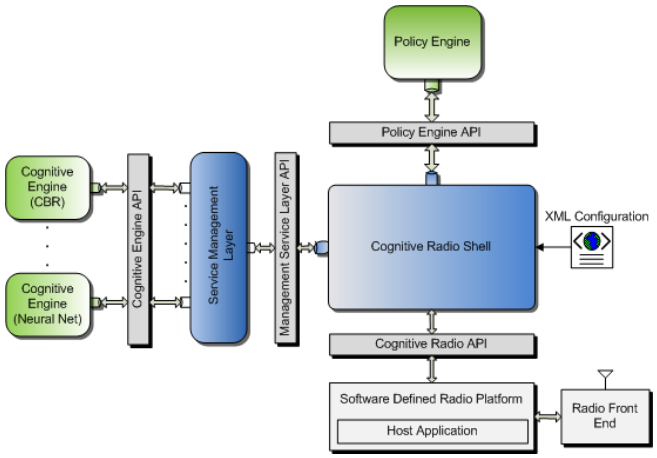

Figure 1. Overall CROSS system block diagram showing all mandatory and optional components and the associated API layers between them.

3.1. Cross Components

The current CROSS component library consists of five categories of components, several of which are optional in a radio system. The five primary components of a CROSS radio system are as follows:

- Cognitive Radio Shell (CRS)
- Cognitive Engine (CE)
- Policy Engine (PE)
- Service Management Layer (SML)
- Software-Defined Radio Host Platform

Figure 1 shows the general CROSS system block diagram, including all mandatory and optional components. The following sections provide more detail on each of the system components.

3.1.1. Software-Defined Radio Host Platform

The SDR host platform is not a component included in the CROSS source code, but it is a necessary component of a CROSS radio system. The host platform is where the client application (sometimes referred to as the 'host application') that calls CROSS library functions is running. Typically, this is also where the radio hardware is located, although this is not a requirement. Without the host application and platform, the CROSS radio system will sit idle without operating instructions.

The most important aspect of the host application is that it works with a specific SDR framework. Within the application running on the SDR, the client code interfaces the CROSS radio system via the CROSS library. The host application can be anything from a standard TCP/IP stack that uses CROSS to optimize network parameters, to a DSA application that is capable of gathering information about current spectrum use and adapting accordingly.

3.1.2. Cognitive Radio Shell

The CRS has several core functions that are integral to component communication and integration within the radio system. It acts as a message passer, configuration parser, and as an interface from the application to the rest of the radio system.

Initially, the CRS parses the Radio XML configuration file that specifies the operating parameters, environmental parameters, and objectives of the current radio application. It passes this parsed information about the capabilities of the radios onto other components in the system such as the various cognitive engines or policy engines that may be connected.

The primary function of the CRS is to act as a gateway between the host application and the rest of the CROSS system. The CROSS library interfaces with the CRS and allows the host applications to issue commands to the system. This single interface makes accessing a CROSS system extremely simple.

3.1.3. Cognitive Engine

Cognitive engines can come in many shapes and sizes. A primary goal in the development of the CROSS framework was to create an architecture that would work with any type of CE implementation. Cognitive engines have been developed with genetic algorithms (GA), case-based reasoning (CBR), and other mathematical models.

Each cognitive engine implementation has its own benefits and weaknesses, and some might be more appropriate for a certain radio system or mission than others. In the event that multiple CEs of different types are connected to the same CROSS radio system, the CROSS radio can selectively use the one that best fits the current environment.

A major difference between the CROSS system and other systems with more integrated cognitive engines is that the host application dictates when CROSS cognitive engines generate new parameters. We do not require the system to constantly generate parameters and push them to the radio. This decision is left up to the host application. Since a CROSS radio system is typically distributed over a network (although this isn't necessary), this keeps network bandwidth open when it isn't needed, and allows for more flexible implementation models.

The host application, using the CROSS library, invokes CROSS system hooks to optimize some set of parameters. The CE returns the optimized parameters set according to its internal.

As long as a cognitive engine has an interface that accepts the CROSS component interface commands as defined by the CROSS API, the internal operation of the CE could be anything.

3.1.4. Policy Engine

The CROSS framework also provides for policy engine components. In general, PEs within the CROSS architecture act as a validation phase for the output of the CE. When a PE connects to the radio system, the CRS will then check all parameter outputs from CEs with the PE.

The PE then determines whether the parameters conform to the active policies and returns a decision array denoting the invalid parameter values (if any) and the reason they were denied. The decision array allows a more fine-tuned approach to policy feedback. Instead of a simple yes-or-no result, the PE informs the system which values were not acceptable and why.

Policy-managed radio is an area of research unto itself, and there exist many sources of further information - e.g. [8] and [9].

As with all other components in the CROSS system, the PE can be implemented any way the developer chooses. It could be a custom PE with a small policy database and simple decision engine, or a PE could implement the XG policy engine and use policies written in OWL. The only requirement is that the PE implements the CROSS component communication API.

3.1.5. Service Management Layer

The Service Management Layer (SML) gives the system the capability to perform complex missions that may depend on numerous smaller services. More complex cognitive radio systems may need to execute several different tasks, with the output of one task determining the next task to execute. These more complex systems can be created using the SML and building an SML XML configuration file that describes the decision models of the SML. We go into much more detail regarding the SML in Section 3.

4. THE SERVICE MANAGEMENT LAYER

The Service Management Layer (SML) is an optional component that is provided with CROSS. When the SML is introduced into a CROSS radio system, it takes control of the system's operation and turns the radio architecture into a Service-Oriented Architecture (SOA). The SOA operating model has a lot of benefits that apply to distributed systems like CROSS.

4.1. Service Oriented Architecture Basics

The basic concept behind a SOA is that a system can be comprised of independent (or very loosely associated) components which each provide a service to the system as a whole. These services can be grouped or reused as necessary to achieve the system's objectives. As long as the different components know how to talk to each other, the components themselves can be completely different in terms of implementation.

The architecture is therefore designed to be distributed and modular. The components can be developed with entirely different models, in different programming languages, for different platforms, and as long as they have a network over which to communicate it doesn't matter where they are located (assuming, of course, that the network has the necessary bandwidth and reliability).

4.2. The SML Missions

In addition to facilitating distributed and modular component design, the SML provides CROSS with another very useful functionality. By configuring SML with 'missions', or radio operating objectives, the SML can delegate tasks to the various components comprising the radio system in pursuit of accomplishing that objective. In effect, the SML is able to use many, perhaps fundamentally different services, to achieve a higher-level radio objective. Each service can be executed based on feedback from the previous service.

During radio operation, the SML is told which pre-configured mission it should pursue. This can be changed on-the-fly via the CROSS system library, as long as the SML has been configured for the desired mission (see the following sub-section).

4.2.1. Configuring the SML Missions

All SML configuration is done via XML files, including configuration of the SML missions. The XML files describe what services are used to accomplish the mission, and what the data flow should be depending on feedback from the various services.

An example of an SML mission configuration can be seen in Figure 2.

```
<mission name="CovertJamEnemy" id="3">
    <services>
        <service name="ClassifyEnemySignals"></service>
            <if value="wifi">
                <service name="DetectWiFiChannels"></service>
                <service name="OptimizeJamWiFi"></service></if>
            <if value="bluetooth">
                <service name="DetectBluetoothDevices"></service>
                <service name="OptimizeJamBluetooth"></service></if>
       <service name="MonitorForReemergedSignals"></service>
    </services>
</mission>
```

Figure 2. Example SML Mission configuration.

In this example, the mission is "CovertJamEnemy". During operation, the first service the radio will use is "ClassifyEnemySignals". The component that provides this particular service returns feedback to the radio, which the SML then uses to decide which service to use next – in this case, the radio takes different actions depending on whether WiFi, Bluetooth, or both were detected.

The SML can be configured with as many missions as the user wishes, as long as the services listed in the missions are provided to the radio by a component. If the necessary services are currently available to the SML, it will avoid that mission functionality.

4.3. SML Operation

The SML can connect and become a part of a CROSS system at any time – the radio does not need to be 'restarted' or 'turned on' with the SML present in order for it to work.

When activated, the SML first reads its configuration file and builds a database of missions and the required services to execute those missions. It then reaches out to the CROSS shell component and registers itself with the radio system.

The CRS then notifies all other CROSS components of the SML's network location, and cedes control of the radio to the SML. The rest of the components then register the services they can provide with the SML, and wait for instructions.

An in-depth example of the radio's operation from this point forward is provided in Section 6 of this paper.

At any point, if the SML is pulled out of the system or becomes unavailable (e.g. if the network becomes disrupted), the CRS will re-assume control of the radio system.

4.4. Requirements of the SML Design

Clearly, if the SML is present, it plays a very central role in the system operation. As such, weaknesses in the SML implementation could severely hinder the radio's operation.

Speed is a top priority for the SML. Since each components' operation is essentially controlled by the SML delegating tasks to it, slow SML operations would bottleneck the radio's operation as a whole. SML database access, decision speed, and message passing must all be fast enough to keep up with the speed required for proper cognitive radio functionality.

In addition, the distributed modularity provided by SOA is useless if developers are not given well-defined APIs and communication interfaces. Without such interfaces, third-party development is impossible, which completely defeats the purpose of CROSS.

5. CROSS APPLICATION PROGRAMMING INTERFACES

The term 'CROSS API' can refer to a number of things: the component communication protocol that the CROSS API defines (via which components receive commands and information and return output), the component API that is used to create new CROSS components, or the CROSS library API which the host application uses to communicate with the CROSS radio system. This section will discuss all of these aspects of the CROSS API.

5.1. Component Communication Interface

All inter-component communication in a CROSS radio system occurs via socket connections over a standard TCP/IP network. The only requirement of any radio component regarding communication with the rest of the radio system is that it implement the CROSS component interface.

Currently, all messages sent to and from components are constructed of pure ASCII. VTCROSS plans to create a more optimized and efficient protocol consisting of specially designed CROSS packets, but this work has not yet been completed.

Each component possesses component-specific registration and de-registration (both for the component and the services each component provides in the event a SML is present). In addition, all components can interpret some standard commands critical to a CROSS radio - such as notification that an SML has joined the radio system and where it is located, or that the entire radio system is being shut down and that the component should cease operation.

Finally, each type of component implements messages specific to its particular duty in the radio system. A CE must be able to receive a parameter set to optimize, just as a PE must be able to receive an optimized

parameter set to validate against active policies.

Communication interfaces for each component are well documented in the code and on the VTCROSS website [5].

5.2. The CROSS Component API

The CROSS component API is something that VTCROSS provides for easy component development, but is not at all necessary to create a new CROSS component. As previously mentioned, as long as a new component properly implements the communication interface described in the previous section, the internal operation of the component are abstracted away from the radio operation.

However, VTCROSS provides a structured object-oriented framework for creating new components should you want to build upon code and functionality we have already provided.

All existing CROSS components exist within a strict class hierarchy, most of which serve as parent classes within the object-oriented class tree. This enables a developer to quickly develop a new component without re-implementing the functionality that already exists in other components.

For example, a cognitive radio researcher, Sally, wants to create a new cognitive engine component with a case-based reasoning (CBR) backend. Luckily for Sally, the CE component is a core component of CROSS, and CROSS provides a CBR class. Sally will create new classes for her cognitive engine and CBR that inherit from the default CROSS CE and CBR, and re-implement a couple of functions for each. Afterwards, her component will be fully operational and ready to connect to a CROSS radio system. All of the other interal functionality and CROSS component interface is provided to her class via the inheritance structure.

For more information, see the numerous teaching references regarding obect-oriented programming, and the CROSS component API on the VTCROSS website [5].

5.3. The CROSS Library API

The CROSS library API is what the host application uses to hook into the CROSS radio system. When CROSS is compiled for the host platform (on which the host application is running), a shared library is generated which can be used from applications on that system. This library defines functions, or 'hooks', that communicate with the CROSS radio via the CRS component.

For example, when an applicatio2n wishes to change the active mission of the SML, it calls a function to that effect from the CROSS library, passing in as an argument the new mission number. The library interprets this, and sends a series of socket communications to the CRS (one telling the system what this message is, another containing the mission number). When the CRS receives these command messages, it will route them to the appropriate component which will respond accordingly.

6. SYSTEM OPERATION

In this section we provide an example that details the operation of a CROSS radio system using the SML component. Dynamic Spectrum Access (DSA) has, over the past few years, become the primary application for cognitive radio systems commercially and in military systems. A DSA system typically observes its environment and determines the appropriate frequency channel to operate on, with the objective of minimizing interference to the primary user and other signals in the nearby frequency domain.

The example we detail consists of two primary cognitive engine jobs, or services, that will be performed each time the host application determines that a change in frequency may be required: 1) Determine the new frequency band, and 2) Determine the optimal parameters for the new frequency band.

Figure 3 shows the block diagram of the CROSS system using our example services. The system consists of two different cognitive engines, each performing a different service. Both are connected to the SML which requests and routes information between them as needed.

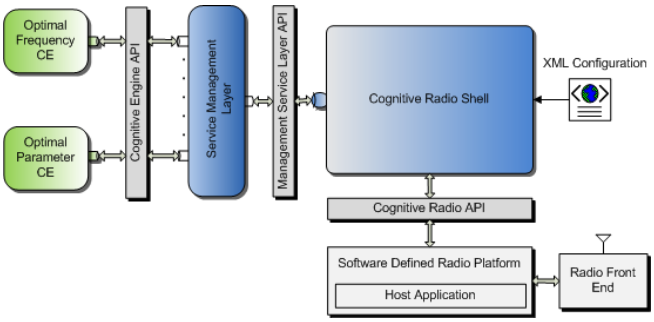

Figure 3. CROSS system block diagram showing two cognitive engines connected to the Service Management Layer component.

The host application sets the active mission that the SML is to perform by calling the *SetActiveMission* function provided by the CROSS C++ library. In our example we only have a single mission - dynamic spectrum access.

Once the CRS receives the *GetOptimalParameters* command from the host application, it forwards the command to the SML. The SML then requests parameter optimization from the CE that has the *OptimalFrequency* service registered by sending it input arguments and the name of the service to perform.

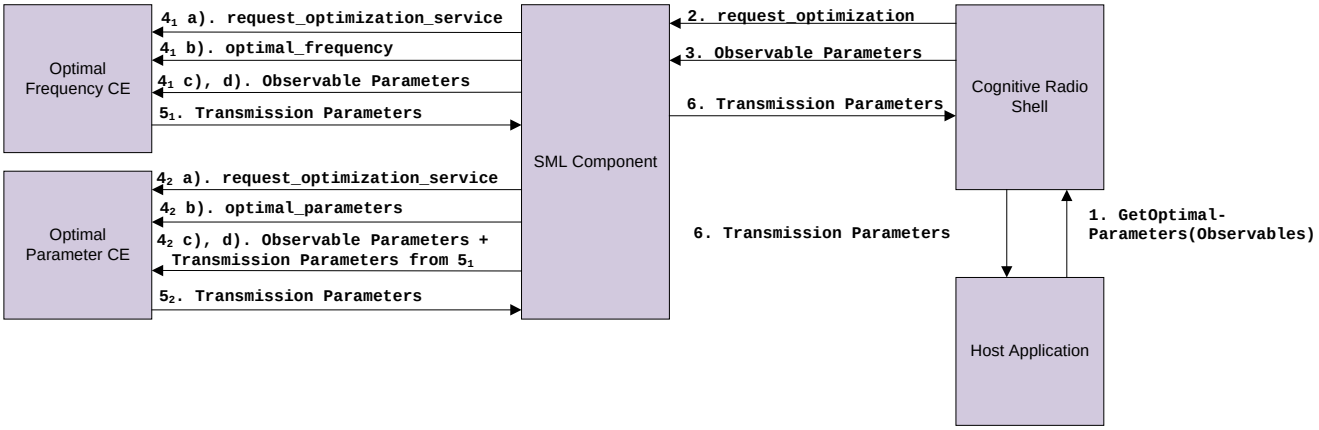

Figure 4. Example optimization protocol flow for the Dynamic Spectrum Access CROSS example with multiple services.

The output of the CE, as defined in the SML configuration file, should be a specific frequency channel. Once the CE provides this output, the SML invokes the *OptimalParameter* service from the CE that registered it, and passes the output of the previous service (the frequency channel) as an input to that CE. The output of the second service and the output of the first service are then passed back to the host application. Figure 4 shows this communication flow visually.

The system described above is able to determine the proper frequency channel to operate on and the optimal parameters for that band. This solution is found in a modular manner, so that multiple components can focus on a single task allowing for a more heterogeneous component environment.

7. CONCLUSION

The VTCROSS project is working to provide a distributed and modular cognitive radio system framework for which it is easy to design, develop, and test radio components, regardless of the target SDR platform. The SML allows for a service-oriented architecture, that further facilitates independent component development and deployment.

Unlike many cognitive radio implementations available today, improvement or adaptations to the CROSS system are not simply limited to one development group or even to one programming language. Indeed, the system is open to any developer to replace any component in any language that supports socket communications (Java, Perl, Python, C++, just to name a few).

To enable this design, VTCROSS provides a well-defined API and component communication interface, which can be used to develop CROSS components in a language of the designer's choosing. In addition, VTCROSS is constructed in an object-oriented class hierarchy, making it simple to inherit from existing classes - further simplifying component development.

The target testbed for VTCROSS is the Virginia Tech COgnitive Radio NEtwork Testbed (VTCORNET), which will consist of 48 SDR nodes deployed throughout a building on the Virginia Tech campus [7]. The combination of VTCROSS and VTCORNET provides a unique system that is openly available for SDR and cognitive radio research and testing.

10. REFERENCES

[1] Open Source SCA Implementation :: Embedded. Available online: http://ossie.wireless.vt.edu/

[2] GNU Radio. Available online: http://gnuradio.org/trac

[3] Harris Corporation Software Defined Radio Solutions, Available online: http://www.govcomm.harris.com/SDR/

[4] Thales Communications Solutions, Available online: http://www.thalescomminc.com/comm_sol.asp

[5] Virginia Tech Cognitive Radio Open Source Systems, https://www.cornet.wireless.vt.edu/trac/wiki/Cross

[6] G.Cafaro et al.,"A 100MHz–2.5GHz Direct Conversion CMOS Transceiver for SDR Applications," in Proc. IEEE Radio Frequency Integrated Circuits (RFIC) Symposium, Honolulu, Hawaii, Jun.2007, pp.189–192.

[7] Virginia Tech Cognitive Radio Network Testbed, https://www.cornet.wireless.vt.edu/trac/

[8] Perich, F., "Policy-based Network Management for NeXt Generation Spectrum Access Control", in IEEE DySPAN, April 2007

[9] Lewis, D., Feeney, K., O'Sullivan, D., "Integrating the Policy Dialectic into Dynamic Spectrum Management," New Frontiers in Dynamic Spectrum Access Networks, 2007. DySPAN 2007. 2nd IEEE International Symposium on , vol., no., pp.390-398, 17-20 April 2007