Adaptive Dynamic Radio Open-source Intelligent Team (ADROIT): Cognitively-controlled Collaboration among SDR Nodes

(Invited Paper)

Gregory D. Troxel*, Eric Blossom‡, Steve Boswell*, Armando Caro*,
Isidro Castineyra*, Alex Colvin*, Tad Dreier¶, Joseph B. Evans†, Nick Goffee*, Karen Zita Haigh*,
Talib Hussain*, Vikas Kawadia*, David Lapsley*, Carl Livadas*, Alberto Medina*,
Joanne Mikkelson*, Gary J. Minden†, Robert Morris§, Craig Partridge*, Vivek Raghunathan*,
Ram Ramanathan*, Cesar Santivanez*, Thomas Schmid¶, Dan Sumorok*,
Mani Srivastava¶, Robert S. Vincent*, David Wiggins*, Alexander M. Wyglinski†, and Sadaf Zahedi¶

*BBN Technologies, Cambridge, MA 02138 USA Email: gdt@bbn.com
†Information and Telecommunication Technology Center, The University of Kansas,
Lawrence, Kansas, 66045 USA Email: gminden@ittc.ukas.edu
‡Blossom Research, LLC, Reno, NV 89506 USA Email: eb@comsec.com
§Computer Science Department, Massachusetts Institute of Technology, Cambridge, MA 02139 Email: rtm@csail.mit.edu
¶Electrical Engineering Department, UCLA, Los Angeles, CA 90095 USA Email: mbs@ee.ucla.edu

***Abstract*— The ADROIT project is building an open-source software-defined data radio, intended to be controlled by cognitive applications. The goal is to create a system that enables teams of radios, where each radio both has its own cognitive controls and the ability to collaborate with other radios, to create cognitive radio teams. The desire to create cognitive radio teams, and the goal of having an open-source system, requires a rich and carefully architected system that provides great flexibility (enabling cognitive applications to change the radio's behavior) and also has a clear structure (both so that others may add or enhance the software, and also so that the system can be clearly modeled for cognitive applications). What follows is a summary of the ADROIT system and the key architectural features intended to enable cognitive radio teams.[1]**

I. Introduction

Software-defined radios (SDRs) have existed for nearly 15 years [1], [2]. About six years ago, researchers came to realize that combining cognition with SDRs was likely to yield far more flexible and powerful radio systems [3]. Yet, as a research community, we are still struggling to develop an understanding of how best to combine cognition and SDRs. Currently, the community lacks either an architecture or a reference implementation that is commonly agreed upon. The problem is even more acute if we focus our attention on SDRs used for data communication. One of the challenges in software-defined data radios is that they require cognition not just within the individual radio, but cognition across the teams of radios that seek to communicate with each other.

The goal of the ADROIT effort is to change this situation. ADROIT is creating open SDRs for data communication and demonstrating how to use cognition to manage teams of these radios. Specifically, we seek to achieve two broad goals:

- *Enable cognitive radio teams.* A cognitive radio team is one where multiple highly-configurable radios can intelligently and dynamically assemble and configure themselves to meet the needs of a particular application or suite of applications. Central to this idea is the notion that applications (or suites of applications, or controllers acting on the applications' behalf) are cognitive and capable of adapting the radios' behavior to best meet the applications' needs.

  Observe that the ADROIT definition of a cognitive radio differs slightly from the classic definition of Mitola and Maguire [3]. In their definition, cognition is internal to the radio. In the ADROIT definition, the applications using the radio (and cognitive controllers) are cognitive and (potentially) self-aware, but the radio itself is not necessarily cognitive. This difference implies that the radio must expose its internal workings such that the applications can manage the radio's behavior.
- *Create an open-source real-time composable software-defined data radio.* SDRs have been in existence for several years. The glory of SDRs is that their behavior is programmable – a radio can be a cell phone or a WiFi hub and the change is a simple matter of programming.

[1]This paper is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under contract number NBCHC050166. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA); or its Contracting Agent, the U.S. Department of the Interior, National Business Center, Acquisition & Property Management Division, Southwest Branch.

In ADROIT, we seek to take these radios to another level of flexibility. First, we want the radio software to be open source. Open-source software is a well-known method to drive innovations, and t if we are to fully exploit the power of SDRs, we need to make SDRs easy to acquire and experiment with. The open-source approach is the best path.

Second, we seek to make the radios composable in real-time. Instead of the prevalent practice of stopping the radio and loading new code to change the radio's performance, we seek, insofar as possible, to allow the radio to change or evolve its behavior while running. Real-time composability is essential for cognition, where the ability to swiftly adapt to changing conditions is required.

Real-time composability also would appear to be a meritorious goal in itself. A radio that can reconfigure in microseconds is, intuitively, far more powerful than one that reconfigures in seconds. Furthermore, a composable radio enables others to replace individual components and use the rest of the system, thereby enabling research while reducing the need to duplicate existing work.

After briefly summarizing the past literature, we use the rest of this paper to explain how we seek to meet these two goals.

II. Prior Work

The ADROIT effort is the beneficiary and outgrowth of two complementary research thrusts. The first is the evolution from software-defined radios (SDRs) to software-defined data radios and the second is the emerging application of cognition to network management and SDRs. We trace both thrusts in the next two subsections.

ADROIT is also the beneficiary of substantial work on open-source systems for radios and data communications, in particular GNU Radio and Click. As ADROIT is enhancing both GNU Radio and Click, it was more useful to present them in conjunction with the enhancements and the discussion of GNU Radio and Click is therefore deferred until section IV.

A. Software-defined Radios and Software-defined Data Radios

The evolution of SDRs has generally followed the evolution in the power of Digital Signal Processors (DSPs).

The first software radios emerged in the early 1990s as it became clear that DSPs had reached a level of sophistication and performance such that they could process the output from (or input to) an analog-to-digital-converter in real time. DSPs with this level of performance permitted radios whose waveforms could be changed by simply altering the DSP's software. The result was a series of radios, beginning with SpeakEasy [2], then the Joint Tactical Radio System (JTRS) [4] and the Vanu radio [5], that realized a modular radio, capable of mimicking the over-the-air behavior of existing radios (e.g. push-to-talk handsets, cellular phones or FM radios) or implementing new waveforms.

As DSPs have become more powerful, and as better hardware developed for frequency selection, the range of potential capabilities of the software radio expanded. It became possible around the year 2000 to imagine radios that dynamically scanned the spectrum, looking for available capacity (e.g., the neXt Generation [XG] radio research project at DARPA [6]), or simply trying to understand how the spectrum was being used (e.g., the WolfPack program at DARPA [7], [8]). The central idea was that the radios would change their behavior based on current spectrum conditions. XG seeks to find and utilize idle spectrum. WolfPack seeks to understand the purposes for which the spectrum is being used.

In the past few years, there has been an additional step. Fast, inexpensive embedded processors has allowed not only the radio frequency and waveform to be programmable, but also the data communications media-access protocol layered on top of the frequency and waveform to be programmable too. The result is a software-defined data radio. The idea is to view data communications protocols such as 802.11 and TCP/IP as fungible protocols, whose behavior is adjusted (or completely reworked) according to current needs. ADROIT takes this idea one step further by viewing the entire protocol stack as fungible; it is assembled in the form needed by the applications currently using the radio.

B. Combining Cognition with Network Management and Software Radios

The other research thread influencing ADROIT is the desire to combine cognition with network management and with software radios.

Network management, at its core, is the process of making sense of a vast amount of information. An individual device may make thousands of variables visible at any time. A single link may see millions of packets per second. Then, once the condition of the network or its components has been assessed, a manager must determine which of thousands of configuration options to adjust in order to improve performance or repair a fault.

This kind of environment is generally viewed as ideal for cognitive tools, which do much better than people at juggling hundreds or thousands of variables. There have been suggestions to embed cognition into network management [9], as well as efforts to allow cognitive entities to mediate between applications and a balky network [10], [11].

There has also been a recognition that cognition is extremely well-suited to network management of SDRs. The XG project found that cognitive tools were essential to allowing the radio to decide which frequencies were free and how to best exploit them. (Part of the issue for XG was that the availability of a frequency was determined not just by what traffic could be found at that frequency, but also by a complex set of FCC rules dictating how the frequency could be used). In an experiment at BBN, we found that a cognitive tool using genetic algorithms was far better at configuring a software radio with over a thousand configuration options than the best-trained radio engineers.

ADROIT is building a software-defined data radio that is intended, from the start, to be cognitively controlled.

C. Terminology

One thing about software radios that has not fully evolved is the terminology. However, some researchers (see, for instance, [12]), have begun to make a distinction between *software-defined radios*, in which most signal processing is done in hardware configured by software, and *software radios*, in which most signal processing is done in software. In this paper, we use the two terms inter-changeably to mean a radio in which most or all signal processing is done in software. We further enhance the concept with a *software-defined data radio* in which both signal processing and higher layer protocols are done mostly or entirely in software.

III. The ADROIT Approach

Reflecting the two research thrusts that have influenced ADROIT, our approach has focused on two central ideas.

First, we set out to make the ADROIT software radio composable. To achieve this goal we first integrated and enhanced two existing open-source systems, GNU Radio and Click, as described in section IV.

The second central idea is ensuring that the radio can be cognitively controlled. In ADROIT, this means that modules of the object may be monitored and adapted in rich, real-time manner. Further, behavior can change within one radio, or across a team. Cognitive control required the creation of several mechanisms to manage the composable software and radio hardware.

First, ADROIT needed a consistent way to model all the components of the radio. This model is described in section V.

Reconfiguration consists of selecting the best components for the current task. (In ADROIT, we use the term "reconfiguration" rather than "configuration" to emphasis that the configuration is dynamically changing). Reconfiguration is managed by a Reconfiguration Manager, described in section VI.

Adaptation is a collective activity, in which (both cognitive and non-cognitive) components of the radio decide how best to tune parameters in the current configuration. Because the tuning is collective, it is essential that all components have a shared view of the radio's operation. Providing that shared view is the job of the Broker, described in section VII.

Finally, because ADROIT seeks to create radio teams, we needed to add some team infrastructure to ensure communication and security. This infrastructure is discussed in section VIII.

IV. Creating a Composable Software-defined Data Radio

Central to ADROIT is the idea that the radio is composable. In the ADROIT context, this means that the radio is made up of a collection of code modules which can be dynamically inserted and removed from the running configuration as needed. Our goal is to make modules represent small units of functionality. For instance, a module might represent a round-trip time estimation routine or a checksum routine.

As we worked through the design of the system, it quickly became clear that modules needed to be typed. For example, if we have two round-trip time estimation routines, one based on mean deviation and the other on standard deviation [13], there needs to be some way to say that both modules take the same inputs and do the same thing, just using a different algorithm. Object classes with inheritance solve this problem nicely.

Having conceptually made all code modules into typed objects, we then needed to solve the problem of how to think about connections between modules. The issue is that the data entering and exiting an object is not only typed, but that only certain types of objects should swap data. In ADROIT, we solved this problem by defining the notion of an object *dependency*, which simply states that to function correctly, an object depends on the presence of certain object types.

Given this model, the next concern was how to implement it in an open-source environment. We could have simply sought to build yet-another open-source system, but our preference was to leverage existing work. In particular, we were committed from the start to working with GNU Radio. GNU Radio already had a functional model very close to this model, with the notion of processing blocks (modules) and typed ports (dependencies) connecting blocks.

Additionally, after some study of the open-source implementations of higher protocol layers it became clear that the Click modular router [14] fit the need. Click implements protocols that sit above GNU Radio and had a compatible architecture (namely an emphasis on small code modules and typed ports, and the ability to insert and remove modules at run time). The next few subsections discuss how we added to these software packages to meet the needs of a composable software-defined data radio.

A. An Overview of GNU Radio

GNU Radio is an extensible free software framework for the creation of software radios. The GNU Radio framework also incorporates software that supports the easy integration of a number of hardware modules so that radio signals may be received from, transmitted to, or exchanged with other GNU Radio-based software radios or conventional radio systems.

GNU Radio uses a modular, block-based architecture with a hybrid Python/C++ programming model. The combination of Python and C++ provides a convenient and high performance platform for developers to use in the development of software radio systems. Functionality that requires CPU-intensive processing is implemented in C++ for high performance, while functionality that involves complex interactions between blocks is implemented in Python [15].

One of the features of the GNU Radio framework is an extensive library of pre-defined and tested functional blocks. These blocks provide signal processing functionality, encapsulate sources and sinks of data, and provide simple type conversions. The blocks are written in C++ and typically have

an automatically generated Python "wrapper" or interface that allows them to be manipulated, connected and utilized in Python. New blocks can easily be added to the block library; indeed the GNU Radio community strongly encourages the addition of new blocks implementing new functionality or improved performance.

GNU Radio processing blocks may be hooked together and run from a Python program. The Python program provides a framework for the processing blocks to communicate via buffers. It also provides a simple scheduler whereby the various processing blocks making up a radio transmitter or receiver are executed sequentially depending on the availability of inputs to the block.

A GNU Radio software radio typically consists of the following elements:

- *Sources*: A GNU Radio software radio will have at least one source. Each source is the head of a processing chain or *flow_graph*. An example of a GNU Radio source is the Universal Software Radio Peripheral (USRP) radio. This is a radio front end that connects to a computer via a USB 2.0 bus. GNU Radio has integrated support for configuring and using the USRP
- *Sinks*: A GNU Radio software radio will have at least one sink. Each sink is the tail of a flow_graph. An example of a sink is a sound card.
- *Flow graphs*: A GNU Radio software radio will have a flow_graph that links together each source and sink pair as well as any intermediate blocks that are required to transform the data stream from a source into a format that is understandable by the sink. For example, converting an FM radio signal that is received by a USRP into an audio signal that can be played through a sound card.
- *Schedulers*: A scheduler is associated with each active flow_graph. Each scheduler is responsible for moving data through its flow_graph. A scheduler iterates through the blocks in a flow_graph, identifies blocks that have sufficient data on their input(s) and sufficient space on their output(s) to be able to process data. It then triggers the processing function for those blocks. The scheduler in the current GNU Radio system relies on a steady stream of data input to the collection of blocks to cause the blocks to run and produce output.

The GNU Radio framework provides an excellent environment to create and run complex signal processing functions, and to connect them to the RF world. GNU Radio has already demonstrated its expressiveness and versatility by rapidly implementing a number of very complicated signal processing programs, such as a High Definition Television (HDTV) receiver. GNU Radio provides a strong foundation for radio development that is enabling academics, industry, and hobbyists to collaborate and innovate effectively.

B. Enhancing GNU Radio

The ADROIT effort, in conjunction with the GNU Radio team, is extending the system to better support data communications. There are only a few proposed extensions to GNU Radio, but they are important for packet radio. The Media Access Control (MAC) layer needs low-latency transmission control – faster than the FIFO processing currently implemented in GNU Radio *flow_graphs*.

The extensions allow a flow to execute to fill a buffer, so that the sample data is pre-computed and ready to go upon receipt of a signal. The extensions implement a signaling mechanism that quickly delivers signals to the processing blocks, either from other blocks or from programs running outside the GNU Radio context. Some MACs require tight timing and time-tagging, and that capability is provided by the proposed extensions. Finally, higher layers like to track and operate on collections of bytes together, and in hierarchies of collections. The network layer thinks of "packets", and the link layer considers "frames", either of which may be composed of multiples of the other. The extensions incorporate the ability to manipulate and tag buffers to meet those needs.

Transmission priority is also of concern to the network and link layers. Quality of Service implementations allow the network to match interfaces with different bit rates and loading to each other, as well as allow higher-priority packets to get through, even though lower-priority packets arrived at the interface earlier. This priority needs to reach down all the way to the transmitter in order to satisfy the latency needs of the network layer. If not accounted for by the physical layer, a large lower-priority packet already in transmission, for example, might occupy the channel, preventing a higher priority packet from being transmitted in a timely fashion.

The extensions extend the GNU Radio stream-based paradigm to allow metadata to be associated with data and transported as discrete *messages* of information. The architecture defines a standard format for the metadata and provides functionality to generate, transport, manipulate and parse this information.

The extensions also include a new type of GNU Radio block. We call this block a *message*-block (or m-block). Information flows into or out of m-blocks as messages that flow into or out of bi-directional m-block typed ports. These messages may communicate data, metadata, control information, status information, signals, or a combination. The m-blocks process any control or signaling information that is sent to them and transform any data using information supplied within the associated metadata. Each port has an associated protocol class that specifies which messages may pass into or out of that port. This port typing ensures that only compatible ports are connected together.

The extensions support the needs of time-knowledgeable, priority-based scheduling required for processing m-blocks, as well as reconcile the interoperation between current GNU Radio flow_graphs and the new m-blocks. This is achieved through the use of m-blocks and a hierarchical, quasi-real-time, hybrid scheduling scheme.

The scheduling algorithm is priority-based. Each m-block is assigned a priority which is equal to the priority of the highest priority message in its input buffer. The scheduler determines which m-block possesses the highest priority and

then dequeues the highest priority message.

C. An Overview of Click

Click is software for a modular router. Its basic architecture is crisply summarized by its designers:

> "A Click *element* represents a unit of router processing. An element represents a conceptually simple computation, such as decrementing an IP packet's time-to-live field, rather than a large, complex computation, such as IP routing. A Click router configuration is a directed graph with elements at the vertices. An edge, or *connection*, between two elements represents a possible path for packet transfer. Every action performed by a Click router's software is encapsulated in an element, from device handling and routing table lookups to queueing and counting packets. The user determines what a Click router does by choosing the elements to be used and the connections among them." [14]

From ADROIT's perspective, this is precisely the modularity of implementation that we seek for upper layer protocols in a software-defined data radio.

The challenge in Click is that Click is designed to implement a router. In one way, that is a good thing. In many types of radio networks (most notably ad-hoc networks and sensor networks), every radio is potentially a router or bridge. Indeed, other researchers have worked to enhance Click's software to support wireless routing [16].

However, Click is structured in the expectation that the mechanics of receiving, transmitting, and, to a large degree, the processing of media layer packets is handled by a piece of hardware, managed by a device driver. In ADROIT, however, that's not true. MAC protocols, in all their richness, are to be modularized just as Click modularizes the Internet Protocol. So ADROIT needed to insert the concept of a software MAC layer into Click.

D. Enhancing Click and Creating a Software MAC Layer

The straightforward approach is to create APIs for a software MAC layer and then adapt Click to use that API. That is broadly the approach ADROIT has taken, with two tweaks.

First, ADROIT defines two APIs: an API for Click to use to talk to the MAC layer, and an API for the MAC layer to talk to the (GNU) Radio. In between the two APIs, implementers can write software for any MAC they wish.

Second, ADROIT implements a model MAC layer. A slight surprise was that the rational platform for such a model MAC layer was Click! It turned out that most of problems of implementing a MAC protocol are similar to those of an internet protocol, and so Click mostly contains the correct programming abstractions.[2]

[2]One incompletely solved issue is real-time actions, such as sending an ACK to an 802.11 DATA packet, an event that must happen in a very narrow time window that may be faster than Click can respond. One solution is to push certain real-time response problems down into the Radio layer. Real-time m-blocks could be supported in the DSPs or even in (run-time programmable) FPGAs.

So, in the end, Click was enhanced two ways. Two new APIs were added, and Click was enhanced to support software MACs.

1) Replacing One API with Two: Click has an interface for devices. Packets to be sent are given to the *ToDevice* element and arriving packets are pushed into the system by the *FromDevice* element. The two elements are the effective API to the device drivers.

ADROIT replaces this model with two APIs. The *MAC-Subnet* API connects the subnet layer (what, in Click, currently sits above the {*From/To*}*Device*) elements to the modular MAC layer. The *Radio* API connects a media access layer (MAC) with the radio channel(s) it needs to transmit and receive on. The Radio API is the boundary between Click and GNU Radio.

Working through the Radio API, a media access protocol, can pass protocol and user data to the radio device for transmission over the air; accept data received by the radio device; obtain information about the operational status of the radio device; and control the operation of the radio device (on a per-packet basis only).

The Radio API is conceptually a tiered interface. There are three tiers:

- *Radio*: The entire radio.
- *Phy*: A Physical layer realization (fully defined in terms of frequency, encoding and modulation). A Phy is viewed as being owned by a particular MAC. Multiple Phys may have the same settings.
- *Frame*: A single logical transmission unit. A frame is transmitted or received over a particular Phy.

Operations may occur at any tier, although certain operations (such as turning the radio on or off) make sense only at particular tiers.

Broadly, operations come in two forms:

- There is a collection of operations to transmit or receive a frame over a particular Phy. This interface is largely intuitive, with one possible surprise: it is possible to temporarily change Phy properties for the lifetime of a frame's transmission. For instance, one way to implement spread spectrum is to specify the transmission frequency separately for each frame.
- There is a collection of operations, largely relevant at the Phy and radio level, to learn about and manage the state of the radio.

The Radio API interface is asychronous. Different operations in the radio may take differing amounts of time, so operations may be completed in an order different from the order in which they are invoked.

The MAC-Subnet API seeks to reflect a conceptual shift that occurs fairly low in the network stack, namely the distinction between media access and subnetwork access. Media access is the process of using the medium (in this case RF) to transmit and receive frames of data. Subnetwork access encompasses the larger problem of routing among a set of (reasonably) homogenous nodes in a network . This distinction is perhaps

most clear when thinking of Ethernet. The media access layer transmits and receives Ethernet frames. The subnetwork layer implements Ethernet bridging.

In its current form, the API is simple. It links one MAC layer to one subnetwork layer. So, for instance, we cannot use a subnetwork layer to bridge between different MAC layers. (The expectation is that this model will be enhanced in later versions of ADROIT).

The MAC layer is responsible for transmitting and receiving frames and for tracking the quality of connectivity to neighbors. The MAC makes information about neighbors available to the Subnet layer. The MAC is also responsible for managing buffer memory and notifying the Subnet layer when buffering is in short supply.

Abstractly, the Subnet side of the API is even simpler. The Subnet layer simply provides two primitives to the MAC layer to help the MAC layer in the transmittal and receipt of messages. In particular, the MAC may ask the Subnet layer which of the radio's neighbors need to acknowledge receipt of a frame, and the MAC may ask if a frame just received is actually for this radio. These primitives enable support for bridging (the radio can accept a frame not addressed to it, and relay the frame on) as well as multicast and anycast addressing (where whether the radio is a member of the multicast or anycast group is relevant as is knowing which neighbors should receive the multicast or anycast transmission).

2) A Modular MAC Layer: Between the MAC-Subnet API and the Radio API lies a modular MAC layer. In principle, one can implement a MAC layer in anyway one wishes between the two APIs. However, since ADROIT seeks to make devising new MAC layers easy, we put a lot of thought into how one might structure a modular MAC layer to encourage ease of modification.

The modularity we chose is shown in Figure 1. A key goal is to accommodate a wide-range of MACs (the ADROIT team expects to support five distinct MACs). Replacing one MAC with another should be possible by rewriting as few modules as possible. A brief description of the function of each module follows.

A brief description of the functions of each module follows. These functions encompass the general function of each module and are open to further development. In addition, as shown in Figure 1, some functions (or groups of functions) of the modules are represented as sub-modules for improved clarity of presentation.

- *Channel Access (CA)*. This module contains functionality required to transmit, receive and generally be the point of contact for the radio device. It interacts with the radio device through the Radio API and is the final/first point of exit/entry of over-the-air frames out-of/into the MAC layer. Received frames are processed and dispatched to the pertinent modules. In general, all controls that need to be done "close to the radio" are candidates for being here.
- *Floor Acquisition and Control (FAC)*. This module contains all of the functions necessary to resolve contention for the channel. It includes sending, receiving and processing control messages (e.g., RTS/CTS/slot-information) and computing deference durations (e.g., NAV/slot allocation). It also controls the nature and duration of floor acquisition, deciding, for instance, whether or not use ACKs, RTS/CTS, or how many DATA packets to send in the burst. It obtains frames from the Reliabilty and Queueing module when necessary (e.g., when floor is acquired) and passes the frame to the CA module for transmission. In sum, this is where all of the smarts for distributed resource allocation of the shared channel resides.
- *Reliability control and Queueing (RQ)*. This module contains all of the functions needed to provide reliable packet delivery, and for doing class based priority queueing of outgoing packets. An alternative design would be to place Reliability and Queueing in separate modules. Because many reliability schemes (e.g., ARQ) store packets in queues, the interfaces between separate Reliability and Queuing modules are fairly rich, and it is difficult to specify a complete yet flexible abstract interface. It is expected that particular MAC implementations will have abstraction boundaries between Reliability and Queueing, and with more experience we may be able to separate these modules. Forward Error Correction (FEC), MAC-layer retransmissions (ARQ) and related functionality are to be placed in this module. Queueing disciplines and head of line unblocking are also included in this module. This module is also charged with creating a frame of a given size upon request from the FAC module.
- *Subnet Interactions (SI)*. This module is responsible for all interactions with the subnet layer. As such, it is the coordinating stop for all calls by the subnet layer. It implements these calls by sending/receiving packets and communicating with other modules to implement certain interfaces. In the reverse direction, it handles all of the

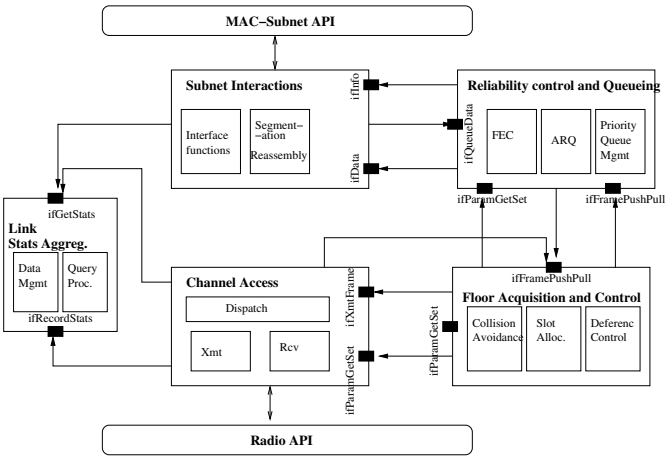

Fig. 1. MAC layer modules and semantic interfaces. Each interface may be supported by multiple implementation-specific primitives that conform to the semantics described in the text. Submodules are shown for example purposes only.

interface calls to the subnet layer on behalf of the other modules. This module also performs segmentation of packets and reassembly, if required.

- *Neighbor Statistics Aggregation (NSA)*. This module collects, aggregates, processes and makes available summary information for use by other modules. It may use sent data and control frames, received data and control frames, or any other activity. Examples include the number of retransmissions per neighbor, the average signal strength of a frame from a given neighbor, the average load or utilization, the average queue length, and error rates. The MAC architecture allows a high degree of flexibility in the amount of processing done on the basic observations – from just aggregating and providing raw data, to smoothing, filtering, estimation and calculating a metric. The NSA module provides an interface through which any module can query for the statistics of a link.

V. Modeling the Radio for Higher Layers: An Informal Object Model

We don't want applications to have to know the innards of GNU Radio and Click to be able to manage or change the radio's behavior. To solve this problem for ADROIT, we created an abstract model of the radio.

Most scenarios seemed best solved by treating the radio and its applications as a collection of objects. Configuration is the best example: if we can think of each software or hardware module as an object with dependencies, building a functional configuration becomes largely a matter of satisfying dependencies across all the objects in a configuration. Similarly, the thousands of variables we need to track for network management are best understood by cognitive applications as attributes of objects, where the objects are part of a type hierarchy. Nonetheless, we observe, there are some parts of the radio, such as low level radio functions, that are reluctant objects.

The ADROIT approach has been to tread lightly. The radio is modeled as a collection of objects. The requirements on objects are quite light, allowing considerable implementation flexibility under an object-oriented veneer.

The basic object model has five components:

- *Type*: A definition of a particular category of module or module that implementers can map their code into. A type is used to characterize a set of services that may be offered by implementations of that type.
  Types use inheritance and types may have more than one parent type. Types may also have more than one implementation or realization in a system.
- *Dependency*: A statement that an object must be connected to another object of a particular type to operate correctly.
- *Parameter*: A visible attribute or member of an object. The parameters of an object are defined by the object's type.
- *Implementation*: A realization of a particular type. A specific piece of code that "implements", or "is a", or "complies with" a Type. An implementation makes the services and parameters (defined by its type) available.
- *Invocation*: Actual running instance of an implementation. More than one invocation of an implementation may be active in a configuration.

This structure neatly solves several problems. For instance, it groups parameters into clusters of well-defined objects. It also expresses the modes in which an object may exist (an abstract type, a realization that can be run, the realization that is current running) and gives us a way (via their type) to quickly identify related implementations or invocations.

VI. Managing Configurations: the Reconfiguration Manager

The role of the Reconfiguration Manager is to create configurations that can be execute on the radio, as well as to start, stop, or change the running configuration, as appropriate. Another useful way to think of the Reconfiguration Manager is that it is the part of the system responsible for controlling how objects transition from implementations to invocations and back.

To perform its functions, the Reconfiguration Manager expects that every object in the ADROIT system will implement the following basic functions (either directly or through a proxy):

- *Run-time control*: An implementation may be started and thus create an invocation. An invocation may be stopped, paused or resumed.
- *Dependency resolution*: Objects know their dependencies and accept instructions as to how they are to be interconnected in the current configuration.
- *State transfer*: In some situations, invocations hold state information that, for the consistency of the radio, must be transferred if the invocations are replaced with new invocations.

VII. The Broker: Enabling Information Flow for Cognition

The Broker serves as a open communications path between objects in the ADROIT system. The idea is that anyone or anything that wishes to observe, monitor, or change the state of an ADROIT radio will do so via a command relayed by the Broker. Furthermore, the Broker will notify interested parties of any changes in the radio's state or configuration. In almost all cases, this involves reading, writing or tracking the value of one or more parameters.

The Broker solves a scaling problem. If there were no Broker, then the addition of a new object could create the need to update every existing object to interface with the new object. This is the classic mxn problem [17] and its solution is to provide a common interface (the Broker), to which every object (new or old) must connect.

Beyond passing commands, the Broker acts as a switchboard and a directory. Entities need know only the name of the invocation whose parameters they wish to update and the

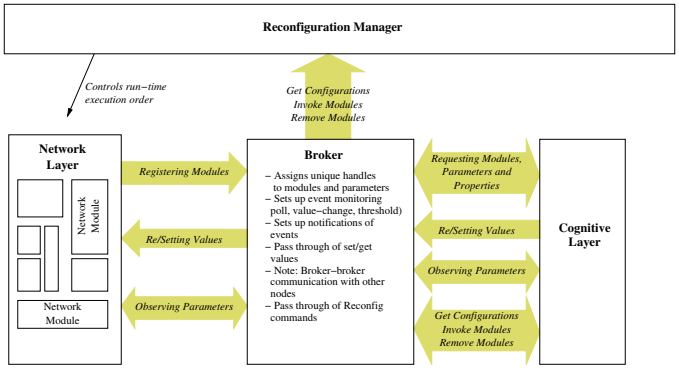

Fig. 2. Broker's role as a system bus, relaying commands and information among its clients.

Broker will find the invocation and make the change. Furthermore, the Broker maintains a directory of all implementations that are available for use, and a copy of the current system configuration (which is maintained by the Reconfiguration Manager).

To perform these services, the Broker implements a relatively rich interface that contains operations to perform the following functions:

- *Directory Services*: Implementations and invocations register themselves with the Broker. The Reconfiguration Manager places a copy of the current system configuration in the Broker. It is possible to ask the Broker to search for active invocations based on their implementation or type. One can also ask for the current configuration, showing how the invocations are interconnected.
- *Parameter Management*: The Broker views every invocation as containing a suite of parameters. These parameters may be read, monitored (e.g., to determine if they change to a value outside expected limits), and some may be altered (to change system behavior). As part of the directory services, it is possible to ask for the list of parameters associated with an invocation.
  The Broker also defines an interface for communication with invocations to read, change, or reset the values of parameters.
- *Configuration Management Pass Through*: The Broker is not responsible for configuration management. However, since so many of the Broker's clients need to know the configuration and also for simplication of APIs, the Broker maintains a pass-through interface, in which requests regarding configuration (including checking the viability of proposed new configurations) are passed through to the Reconfiguration Manager.

VIII. Infrastructure for Radio Teams

Given modular software, a Reconfiguration Manager to manage the radio's configuration, and a Broker to ensure coordination, we have all the components needed to build an agile, cognitive software-defined data radio. However, ADROIT's goals go beyond enabling single radios. We seek to build (cognitive) radio teams. To enable radio teams, we need to add two more mechanisms: a coordination channel and a security model.

A. A Coordination Channel

Suppose we turn on a group of radios in a region. How do the radios find each other? How do they coordinate?

It would be nice to imagine that the radios could dynamically search the spectrum and find each other. But, so far, no solution exists for the dynamic discovery problem. So ADROIT does what everyone else does: it defines a small bit of dedicated spectrum to use as the coordination channel (in ADROIT, we call it the *orderwire*).

Radios discover each other via the orderwire and are then free to use the orderwire to negotiate to use other frequencies for communication. Even if another communication frequency has been negotiated, radios must periodically check the orderwire for newly arrived radios that wish to join the team.

It may seem wasteful to permanently allocate spectrum to the orderwire. Studies suggest, however, that if the orderwire is used wisely, the improved spectrum utilization that SDRs can achieve by coordinating usage outweighs the loss of bandwidth due to the allocation of the order wire [18].

B. Security Model

Just because a radio begins to use the local orderwire does not mean that the radio should be permitted to join a given radio group. How does a radio group distinguish between radios that it should admit to the group and those it should not? And how does the team protect itself against hostile radios?

ADROIT (at least for the moment) has decided to focus on problems of radios that seek to join teams they are not authorized to join and radios that successfully join and then seek to subvert a team. ADROIT does not seek to address cases of radios that seek to jam or perform denial of service style attacks on communications channels.

The ADROIT security model is simple. Every radio carries signed public key certificates from one or more authorization authorities. *Initiators* are radios whose certificate's attributes includes the ability to define a team, where a team is defined by the characteristics of the radios allowed to join it. So, a radio can join a team if it has a certificate issued by the team's initiator; admission can also be contingent on holding other, pre-existing attribute certificates as defined by per-team policy. Furthermore, members of a team may define their own multicast groups within the team, with admission defined by a separate subteam roster. Thus communications, even within the team, can be kept to a "need to know" subgroup.

Currently ADROIT focuses on protecting IP-layer communication (both multicast and unicast) by combining the above admission controls with IPsec. Later, we will consider security extensions for our subnet-layer routing protocol based on the same model.

IX. Cognitive Teams

Having discussed all the components of an individual ADROIT radio, it is now time to look at cognition and how

ADROIT enables cognitive teams. We should emphasize that this section, in particular, discusses work in (early) progress. Creating cognitive radio teams in the style ADROIT envisions is very much a research challenge.

A. The Cognitive Layer on Each Node

ADROIT explicitly seeks not to favor one mode of cognition over another and, indeed, seeks to permit multiple modes of cognition to coexist on a single node. As part of this mindset, ADROIT thinks of cognition not as being resident in a particular entity or application but rather in a *cognitive layer* where multiple cognitive applications may co-exist.

The implication of this design is that ADROIT radio teams will be heterogeneous in cognition. Even if the radios are all running ADROIT software on the same hardware, their cognitive layers may contain different mixes of cognitive applications.

B. Multi-Node Coordination

Because the (independent) cognitive layers of different nodes will be changing parameters and network configurations dynamically, a very real risk is that applications will be unable to communicate. The different applications will have certain communication requirements and expectations they impose upon the broader network – such as the receive frequency, the transmit frequency, the expected header format, the maximum header size, the bit-representation used (4 bit, 16 bit, etc), the re-send protocol, the backoff protocol, the acknowledgment protocol, the unicast/broadcast assumption, quality of service needs (e.g., projecting forward), and reservations (block off resources). These requirements may conflict, both within a node (where they will presumably be swiftly detected) and also between nodes (where multi-node coordination will be required).

As a result, each ADROIT node will have a *Coordination Manager* responsible for maintaining inter-node coordination. Coordination is the act of managing interdependencies between activities [19].

In this environment, the cognitive layer must reason about what is currently being applied within the node, assess the likelihood that it will significantly impact neighbouring nodes, and then ask the Coordination Manager to manage the interdependencies. The Coordination Manager must also be able to robustly handle the change over. No change in protocol, module, or parameter setting should cause significant long-term adverse effects in the network.

Note that coordination may be regional; i.e., it does not need to apply to the entire network. Regions may be defined in different ways, including geographically, by task, or by organizational hierarchies. For example, all sensors may be communicating on Channel A, while all people are communicating on Channel B. A possible network might include one or more nodes that serve as communication bridges, running multiple protocols (one for each region).

One consideration is that that orderwire bandwidth will be limited, and may have to be shared among a number of radio teams. So, as a starting point, our assumption is that we would like to keep coordination traffic between Coordination Managers modest.

One approach to minimizing cross-node communication is to *bookmark* safe states. That is, the cognitive layer tracks the performance of the node and network, and keeps a list of previously experienced working configurations (potentially sorted by their successfulness). When the cognitive layer changes a parameter that causes the network to stop functioning correctly (or to dramatically reduce performance), it can return to the bookmarked state. If a bookmarked state no longer performs well, then it is likely that other nodes have changed their configurations for unsafe values, or that the environment has changed dramatically. In these cases, the Coordination Manager can resort to the orderwire to re-coordinate, perhaps asking all nodes to return to a bookmarked state.

An alternate approach is to support inter-node *negotiation*. Negotiation is the communication process used by a group of agents in order to reach a mutually accepted agreement on some matter [20], and is a vibrant area of Artificial Intelligence research [21], [22], [23]. We assume that the Coordination Manager will negotiate over the orderwire unless alternative communication regions can be easily identified. A research issue lies in deciding what terms to negotiate.

A third approach is to develop a *handshake* mechanism (deadlock-free) that allows two different nodes to synchronize changes.

Duplication approaches may also be appropriate for certain classes of changes, in which a change results in both protocols being applied until it is clear that both sides have changed protocols. This approach may apply in some cases (e.g., a change in header size), but not in others (e.g., for transmit/receive frequencies).

It is currently undecided whether a node will make the decision locally and then coordinate with its neighbours, or whether the decision will be jointly made.

C. Information Sharing

In addition to basic coordination, nodes may need to (or desire to) share information. For example, if two nodes have explored different parts of the environment (either geographical or communications), they may wish to share their observations. We want to have the ability for a cognitive layer to be able to receive information from multiple nodes, identify appropriate patterns and issue appropriate reconfigurations – both within and across nodes, as needed. A key research interest is to identify meaningful patterns of behaviours across the network and choose the right response for multiple nodes.

The Coordination Manager will decide *what* information to share across nodes, and *when* to share it. Information could include current configurations, local observations, or current models. To calculate what information to share, we will use an estimate of *information benefit* that manages temporal decay, context- and task-sensitive importance of the data, and monitors the effect of sharing information on overall

performance, as in [21], [24], [25]. To calculate the *cost of information*, we will monitor the additional overhead caused by information sharing, and estimate cost proportional to available resources. The Coordination Manager will combine these two measures to effectively evaluate the *utility of information sharing*, thereby sharing relevant information when network resources are available.

The Coordination Manager may also decide which module(s) are to perform particular computations. For example:

- The Coordination Manager could select one node to compute the region-wide Quality of Service and select a common communication frequency for all nodes in that region. In this situation, all nodes must send their events to the selected node for processing. This approach is generally easy to implement, may reduce communications, but may be less fault-tolerant.
- The Coordination Manager could choose to distribute a task over multiple nodes so that all cognitive layers take shared responsibility for observing patterns and making decisions regarding that task. While more robust to node and communication failures, it may be difficult to decompose the task.

X. Conclusion

ADROIT is an on-going project. Furthermore, as an open-source effort, ADROIT expects many of its features to evolve and mutate as new parties contribute their code and ideas to the effort. So any conclusions are necessarily preliminary. With that warning having been given, there are some useful observations.

First, one of the challenges is finding multi-layered structured environments. For instance, we want the network code to contain lots of small modules to perform functions such as decrementing a TTL, or computing a CRC. Yet, as the discussion of the MAC sub-layer shows, we also want to clump those modules into larger units of operation. Similarly, in GNU radio we found the need to create *m-blocks* to handle aggregations of data. And to avoid the disaster of thousands of parameters in a flat space, we placed them in the context of a typed object system with inheritance.

Second, creating an environment for cognition is hard. Each radio, much less each radio team, contains a vast amount of data about its performance and a range of configuration options. Making sense of that information, and in a way that multiple cognitive entities can manage is hard. Our approach was to make objects typed, and to create helper applications such as the Broker and the Coordination Manager and the Reconfiguration Manager. Time will tell if this is the right approach.

Acknowledgment

The authors would like to thank Jonathan Smith and Lee Badger of DARPA for their support and insights.

References

[1] J. Mitola, "The software radio architecture," *IEEE Commun. Mag.*, vol. 33, pp. 26–33, May 1995.
[2] R. Lackey and D. Upmal, "Speakeasy: The military software radio," *IEEE Commun. Mag.*, vol. 33, pp. 56–61, May 1995.
[3] J. Mitola and G. Maguire, "Cognitive radio: Making software radios more personal," *IEEE Personal Commun. Mag.*, vol. 6, pp. 13–18, Aug 1999.
[4] "Joint program executive office, joint tactical radio system," SPAWAR. [Online]. Available: http://enterprise.spawar.navy.mil/
[5] "Vanu radio," Vanu. [Online]. Available: http://vanu.com
[6] H. Kenyon, "Smart radios juggle spectrum," *Signal*, Dec 2003.
[7] D. Cousins, C. Partridge, K. Bongiovanni, A. Jacksons, R. Krishnan, T. Saxena, and W. Strayer, "Understanding encrypted networks through signal and systems analaysis of traffic timing," in *Proc. 2003 IEEE Aerospace Conference*, Mar 2003.
[8] "Wolfpack," DARPA. [Online]. Available: http://www.darpa.mil/ato/programs/WolfPack/index.htm
[9] D. Clark, C. Partridge, J. Ramming, and J. Wroclawski, "A knowledge plane for the internet," in *Proc. ACM SIGCOMM Conference*, Aug 2003.
[10] "Sapient: Situation aware protocols in edge network technologies," DARPA. [Online]. Available: http://www.schafertmd.com/sapient/index.html
[11] K. Z. Haigh, S. Varadarajan, and C. Y. Tang, "Automatic learning-based manet cross-layer parameter configuration," in *Workshop on Wireless Ad hoc and Sensor Networks (WWASN2006)*, Lisbon, Portugal, 2006, to appear.
[12] J. Reed, *Software Radio: A Modern Approach to Radio Engineering*. Prentice Hall, 2002.
[13] V. Jacobson, "Congestion avoidance and control," in *Proc. ACM SIGCOMM Conference*, Aug 1988, pp. 314–329.
[14] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek, "The click modular router," *ACM Trans. on Computer Systems*, vol. 18, no. 3, pp. 263–297, Aug 2000.
[15] G. van Rossum and F. Drake, *The Python Language Reference Manual*. Network Theory Ltd., 2003.
[16] B. Chambers, "The grid roofnet: A rooftop ad hoc wireless network," Master's thesis, MIT, Jun 2002. [Online]. Available: http://pdos.csail.mit.edu/papers/grid:bac-meng.pdf
[17] M. Padlipsky, "A perspective on the arpanet reference model," in *Proc. IEEE INFOCOM '83*, 1983.
[18] C. Santivanez, R. Ramanathan, C. Partridge, R. Krishnan, M. Condell, and S. Polit, "Opportunistic spectrum access: Challenges, architecture, protocols," in *Proc. 2nd Annual International Wireless Internet Conference (WICON)*, Aug 2006.
[19] T. W. Malone and K. Crowston, "The interdisciplinary study of coordination," *ACM Computing Surveys*, vol. 26, no. 1, pp. 87–119, March 1994.
[20] S. Bussmann and J. Muller, "A negotiation framework for co-operating agents," in *Proc Cooperating Knowledge-Based Systems (CKBS-SIG)*, S. M. Dean, Ed., University of Keele, 1992, pp. 1–17.
[21] A. H. Bond and L. Gasser, "An analysis of problems and research in distributed artificial intelligence," in *Readings in Distributed Artificial Intelligence*, A. H. Bond and L. Gasser, Eds. (San Mateo, CA: Morgan Kaufmann), 1988, pp. 3–35.
[22] S. Green, L. Hurst, B. Nangle, P. Cunningham, F. Somers, and R. Evans, "Software agents: A review," Department of Computer Science, Trinity College Dublin, Tech. Rep. TCS-CS-1997-06, 1997, https://www.cs.tcd.ie/research_groups/aig/iag/toplevel2.html.
[23] X. Zhang, V. Lesser, and S. Abdallah, "Efficient Management of Multi-Linked Negotiation Based on a Formalized Model," *Autonomous Agents and Multi-Agent Systems*, vol. 10, no. 2, pp. 165–205, 2005. [Online]. Available: http://mas.cs.umass.edu/paper/384
[24] L. Gasser and B. Stvilia, "A new theory of information quality," Graduate School of Library and Information Science, University of Illinois at Urbana-Champaign, Tech. Rep. ISRN UIUCLIS–2001/1+AMAS, 2001.
[25] S. Sen and P. P. Kar, "Sharing a concept," in *Working Notes of the AAAI Spring Symposium on Collaborative Learning Agents*, Mar. 2002, aAAI Tech Report SS-02-02.