AirStore User's Guide

Author: Randall B. Smith

Date: June 19, 2009.

Release: 5.0 (aka: Red)

Introduction

AirStore is a shared data repository for Sun SPOTs and Sun SPOT host applications. With simple one-line puts and gets, distributed applications can share primitive Java data types. One Sun SPOT application can set a variable “x” with

```
AirStore.put(“x”, 17);
```

and another can get “x” with

```
int x = AirStore.getInt(“x”);
```

The data types supported are: `int, double, String, boolean, byte, long`, plus arrays of these types.

AirStore originated from a desire to replace the verbose and oft-repeated patterns of radio usage, which involves setting up connections, creating radiograms or streams, and creating threads that hang on a read to perform some callback when data arrives. The repeated appearance of the same chunks of code in many applications suggested there may be some simpler abstraction waiting to be articulated. Simplifying such code makes it easier to understand and maintain, and we hope the system will be easy to learn and use.

AirStore is implemented as a fully-replicated data store. Each participating Sun SPOT (and the host, if a participating host app is running) maintains a full copy of the data. Because of this complete replication, a `put()` can be broadcast, and a `get()` requires no use of the radio at all. The resulting spare use of the radio can help make AirStore appropriate for sensor network applications.

Current Status

This is the first release of AirStore. Although functional and we believe sound enough for use, there are extensions we hope to add. Providing a “transacted take” mechanism and “dynamic self repair” are high on the list. A transacted take is, as you might guess, a way to retrieve and remove a record atomically with a guarantee that you are the only one to succeed in doing so. Dynamic self repair is mentioned in the next section, and may be important because AirStore is based on unacknowledged broadcast packets.

AirStore Semantics: Putting, Getting, and Taking Records

When it comes to retrieving data, AirStore acts as a kind of *tuple space*[1], as illustrated by Linda[2] and Java Spaces[3]. In any system with no remote object references, such as AirStore and other tuple spaces, the user needs a way to retrieve data. In a tuple space, retrieval is done by a matching mechanism.

1 http://en.wikipedia.org/wiki/Tuple_space
2 http://en.wikipedia.org/wiki/Linda_(coordination_language)
3 http://en.wikipedia.org/wiki/Tuple_space#JavaSpaces

In this section we describe the basic unit of AirStore data, the Record. We also describe the RecordTemplate (used for matching), the AirStore methods for putting Records, and the matching used in getting and taking Records.

Records: The one line code samples illustrated in the Introduction are from an API that is a layer on top of the basic AirStore story. AirStore actually contains an (unsorted) collection of *Records*. A Record object is a collection of key-value pairs, in which each key is a String, and the value is one of the supported data types. So you might think of a Record as a hashtable or dictionary.

Each key-value pair is an instance of RecordEntry. So a Record is simply a collection of RecordEntry instances. There are many subclasses of RecordEntry, one for each of the various types of values, such as int, String, etc. However, normally you do not have to worry about RecordEntry instances, as they are created for you as necessary. Here is an example of creating, filling in, and storing a record

```
Record r = new Record();
r.set(“x”, 17);
r.set(“version”, 1);
r.set(“source”, “0014.4F01.0000.1234”);
r.set(“isCallibrated”, true);
AirStore.put(r);
```

Each `set()` method creates an appropriate RecordEntry subclass instance and adds it to the Record. Note that AirStore does not need to be set up or initialized. It takes care of all that “lazily” upon first use. Consequently, there may be a delay upon first use, as your instance of AirStore announces itself “to the air” and gets updated by some other participant.

In order to keep from “filling up,” AirStore implements a Record Replacement Rule:

> **Record Replacement Rule:** When a record R arrives, AirStore searches itself for a record whose keys match R's keys exactly in name and number. If such a match is found, AirStore removes that older record, and replaces it with R.

Thus if an application repeatedly sends the same bit of data (a Record with the same keys) over and over, AirStore will automatically replace the older Record with the new arrival. If you wish to make a record persist, simply add a key that is unique. If one key is a String that is an integer incremented at each send, you effectively have a version number and the Records will all be retained.

Getting a Record – RecordTemplates: In the opening example illustrating the call `AirStore.getInt(“x”)` we saw an example of retrieving a Record. The `getInt()` call is part of a convenience API which is based on the more general retrieval mechanism, one that works by matching against a template. The `getInt(“x”)` method actually creates a RecordTemplate object to get matches from AirStore, casting the first returned result to an `int`.

Matching can be thought of as a kind of filtering operation over the contents of AirStore. Records are retrieved by matching against a RecordTemplate. A RecordTemplate is a subclass of Record. Here are examples of using a RecordTemplate for retrieval.

```
RecordTemplate t = new RecordTemplate(“x”);
t.add(“y”);
Vector v = AirStore.getAllMatches(t);
```

the resulting Vector `v` contains all records with *at least* an`"x"` key and a `"y"` key. Each matching Record may have other keys as well.

Here is an example that requires a particular value for the "`source`" field:

```
RecordTemplate t = new RecordTemplate("x");
t.add("y");
t.set("source", "0014.4F01.0000.1234");
Vector v = AirStore.getAllMatches(t);
```

The result in the second example is a subset of the results in the first example. The Records in the second example have `"x"` and `"y"` keys, but also contain a RecordEntry with key `"source"` and value `"0014.4F01.0000.1234"`.

The basic call for record retrieval is the AirStore method `getAllMatches( ...)` in which a RecordTemplate object is passed as a singe argument, and a Vector of (a copy of) the matching records is returned. The original records remain in AirStore. `AirStore.getMatch(..)` maybe used, it returns the first match found. Again, there is an API layered on top of this mechanism that automatically creates a RecordTemplate for you to cover the common cases we have already seen, such as `AirStore.getInt("x")`, `AirStore.getString("name")`, and so forth. These `get` methods create an appropriate RecordTemplate, and return the first Record found among the resulting matches.

Here are the rules for matching a RecordTemplate against a Record.

> **RecordTemplate Matching Rule:** A RecordTemplate T matches a record R if each of the RecordEntries in T match some RecordEntry in R.

Note that the match is kind of "generous" in that a template with a single entry may match many records.

Because a RecordTemplate is a subclass of Record, it is necessarily a collection of RecordEntry objects. Often, a RecordTemplate contains RecordEntry objects that are instances of RecordEntryAny. A RecordEntryAny has a key but no value and special matching rules. A RecordEntryAny is automatically added to your template in the above examples when you invoke the `add("y")` method.

> **RecordEntry Matching Rule:** RecordEntry objects E1 and E2 match if the keys and values match. Values of different types never match (The values 1 and 1.0 do not match, nor does the byte 1 match the int 1.) An important exception is the class RecordEntryAny, which is only held by RecordTemplate objects. Instances of RecordEntryAny will match a RecordEntry if the keys match.

Taking a Record. The operation

```
Vector v = AirStore.takeAllMatches(aRecordTemplate);
```

enables you to get matches for a template and simultaneously remove them from AirStore. Unlike a `get()`, a take operation does use the radio, in order to inform all the replicas of the removal. Obviously this is a way to delete records from AirStore, but it can also be used as a kind of coordination mechanism in the style of blackboard systems (http://en.wikipedia.org/wiki/Blackboard_(computing)). Currently, there is no guarantee that if you take a record, you are the only one to have done so: some other application somewhere may have almost simultaneously done the same thing. We hope to add a "transacted take" feature in the future.

Listen/Notify pattern. AirStore supports listening for the arrival, removal, and replacement of a record matching a given template. If you implement the interface IAirStoreListener, you will need to implement three methods:

```
public void notifyPut(Record r);
public void notifyTake(Record r);
public void notifyReplace(Record out, Record in);
```

You express your interest in notifications by executing

```
AirStore.addListener(this, aRecordTemplate);
```

If you are interested in many different templates, call this as many times as you like with a different RecordTemplate each time. These notifications do not involve the radio because of AirStore's fully replicated implementation.

Unless you are particularly interested in the act of replacement, you will want to implement `notifyReplace` as

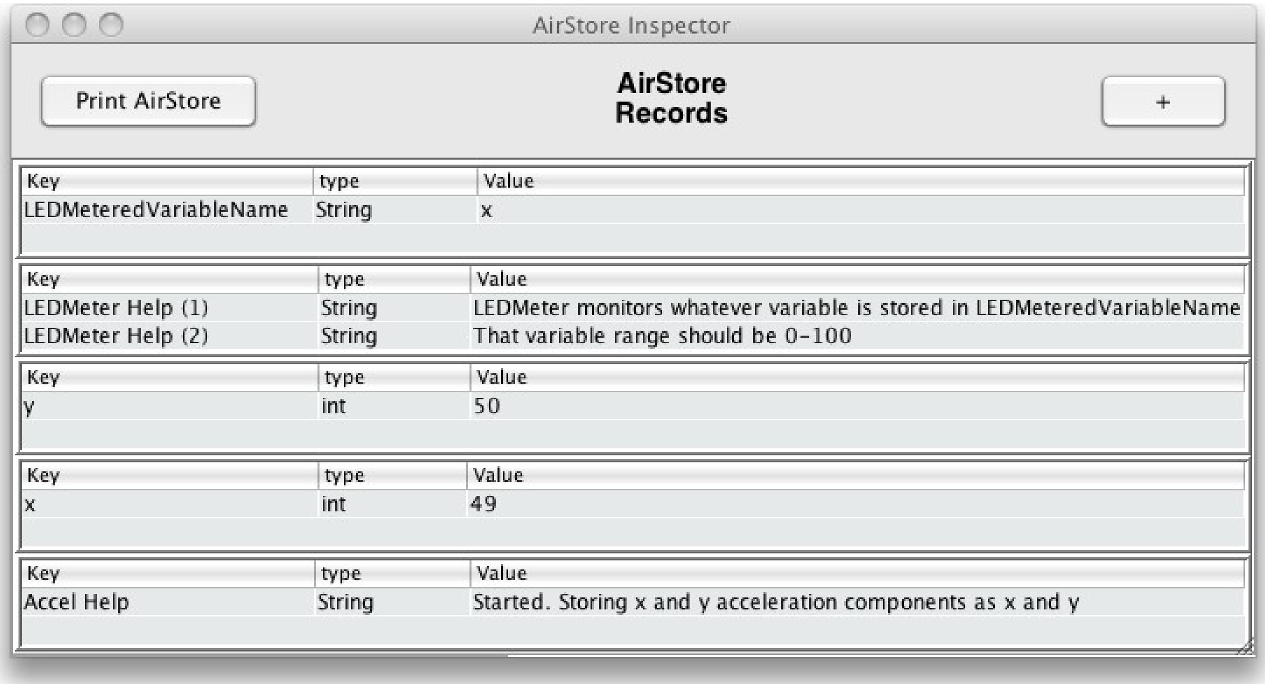
```
public void notifyReplace(Record out, Record in){
    notifyPut(in);
}
```

because, when a record is put and happens to cause a replacement, only `notifyReplace()` is called.

The AirStoreInspector

The AirStoreInspector application runs on the host and shows the current state of AirStore. It also lets you remove or add records through the GUI.

To run the AirStoreInspector, use a shell tool to change into the directory sdk/Demos/AirStore/AirStoreInspector and type "ant host-run," or use NetBeans to open up the directory as a project, and select the menu item "run."



Key	type	Value
LEDMeteredVariableName	String	x

Key	type	Value
LEDMeter Help (1)	String	LEDMeter monitors whatever variable is stored in LEDMeteredVariableName
LEDMeter Help (2)	String	That variable range should be 0-100

Key	type	Value
y	int	50

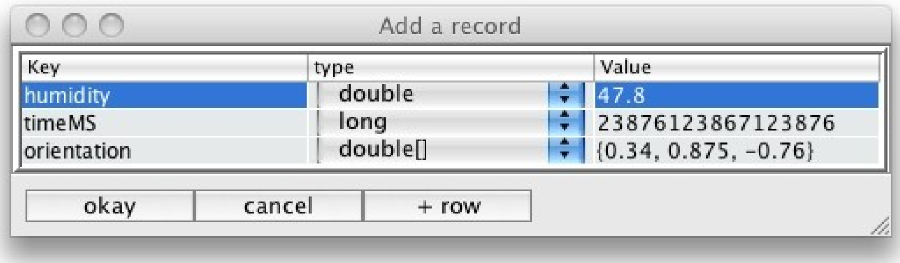
Key	type	Value
x	int	49

Key	type	Value
Accel Help	String	Started. Storing x and y acceleration components as x and y

In the above example, you'll see 5 Records. Each Record consists of a single RecordEntry except for the second Record, which contains two RecordEntry objects.

A “right click” popup menu is available on each record to let you remove that record, or edit a copy to be added to AirStore. The “+” button lets' you add a new Record.

Below is the RecordEditor, which is invoked when you click the “+” button on the inspector, or when you select “edit and add a copy...” from the right-click popup on a record.



Key	type	Value
humidity	double	47.8
timeMS	long	23876123867123876
orientation	double[]	{0.34, 0.875, -0.76}

Note the presence of empty rows in the editor has no effect on the resulting record: any empties are ignored.

The Assumption of a Mutually-Connected Broadcast Group

AirStore assumes all participants are within radio broadcast range of each other. Because broadcast packets are not acknowledged, AirStore itself is fundamentally unreliable. In the future, AirStore may include mechanisms with which a participant can discover if and when it is out of sync with others so it can repair itself. For now, we can only advise you to be aware of this assumption. Of course, many applications work well in the face of dropped packets. Also, in practice, two SPOTs that are the only SPOTs around normally do not miss each other's broadcasts if they are reasonably proximate.

> **Important note on broadcast performance.** Each broadcast packet has a hopcount: When the broadcast hopcount is 2 or more, each SPOT in the network will repeat any broadcast packets it receives, decrementing the hopcount in the repeated packet. Repeating (also called *forwarding*) obviously can increase the reach of your broadcast group. (The default hopcount is 2 for for complex reasons having to do with the basestation and host application addresses.) The current release (5.0, a.k.a. Red) effectively limits the total network broadcast packet forwarding rate to about 14 packets per second. Hence it is important to keep the total broadcast rate in your network below 14 packets per second. If the combined rate exceeds this for an extended period, the forwarding queues will backup and you may experience delays. There is a more serious problem however: a node, seeing a forwarded packet, must decide if this is a new packet, or one that it has already seen. Long queues can cause the node to become confused as to what is a new packet because the 802.15.4 spec allows only 1 byte as a sequence number. With only 256 possible packet IDs and long processing queues, the task of making the new vs. old distinction becomes a matter of artful guessing. Correctly making such a guess every time in all situations is fundamentally a hopeless task, so your application may be handed the same “new” packet multiple times when the total broadcast rate overwhelms the forwarding queue rate. A solution for Air Store would be to provide its own sequence numbering scheme using 32 bits per number, but this is not yet implemented.

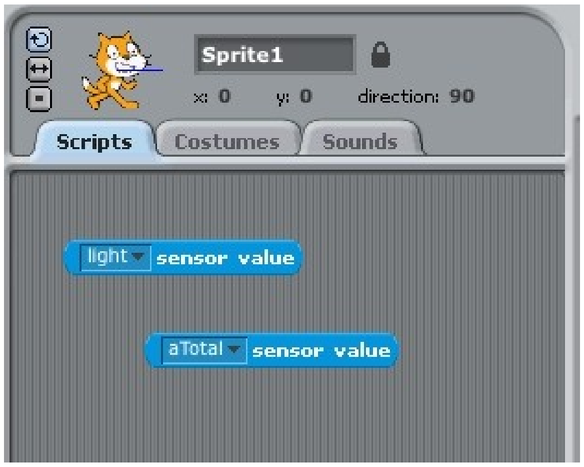
AirStore Demos

Accel This demo (in .../sdk/Demos/AirStore/AirStore_Demos_Accel/) simply reads the accelerometer's x and y components, and puts them into AirStore as values x and y. Check out the source code, it is only a few lines long. Deploy this app to a Sun SPOT and start it running. You may wish to run the AirStoreInspector first so you can see the values as they appear and change when you tilt the SPOT.

LEDMeter This demo (in .../sdk/Demos/AirStore/AirStore_Demos_LEDMeter/) displays a variable value in the range 0-100 on the 8 on-board SPOT LEDs. A bar of red lights of varying length appears. Which variable does it monitor? The answer is itself an AirStore variable named "LEDMeteredVariableName". When this app starts, LEDMeteredVariableName is set to "x", so the meter will show the current "x" value, whatever that is. If the above Accel demo is still running, it will be the x component of that SPOT's accelerometer, which changes when you tilt the spot left or right.

Try using the AirStoreInspector to change the LEDMeteredVariableName from "x" to "y". You should then find that your Accel demo SPOT will change the LED meter when it is tilted away or towards you.

Scratch Scratch (http://scratch.mit.edu/) is a visual programming environment created at the MIT Media Lab – it is aimed at novice programmers. AirStore can be integrated with the Scratch environment through Scratch's sensor variable mechanism. Thanks to Andrew Davison for making three Java classses available (Scratch.java, ScratchMessage.java, and StringToks.java).



The image above shows two Scratch sensor values, representing the light level and total acceleration on the SPOT running the "OnSPOT" portion of this demo.

Here is how to use the demo: In the Scratch environment, enable remote sensors from the right-click popup on any sensor value object, selecting "enable remote sensor connections." Once you do that, run the host app AirStore_Demos_Scratch_OnDesktop . This will connect AirStore with the remote sensor values of Scratch. Deploy and run the AirStore_Demos_Scratch_OnSPOT application on some SPOT. It will export several of its SPOT sensor values for your use within Scratch. When you make a sensor value object in Scratch, the pull down menu will list the variables from the SPOT. These variables can be used in your Scratch applications.

It is also possible to control AirStore variables from within Scratch. Currently, the only variable set up for this use is "ledX".

In the Scratch environment, select the “Variables” pane at left, then press the “Make a variable” button. If you make the name of the variable be “ledX” you can control the position of the lit LED on the SPOT. Double click the variable object to make a slider appear, or type into the variable's text box.

By investigating the source code for this demo, you should be able to make your own extensions to this integration.

Adding AirStore to Your Application

To enable your codde to access the AirStore libraries, you need to modify your project's build.properties file. Add these two lines:

```
user.classpath=${sunspot.home}/Demos/AirStore/lib/AirStore_common.jar:
${sunspot.home}/Demos/AirStore/lib/AirStore_device.jar

utility.jars=${sunspot.home}/Demos/AirStore/lib/AirStore_common.jar:${
sunspot.home}/Demos/AirStore/lib/AirStore_device.jar
```

For a host application, modify the `user.classpath` variable in your build.proeprties file

```
user.classpath=${sunspot.home}/Demos/AirStore/lib/AirStore_common.jar:
${sunspot.home}/Demos/AirStore/lib/AirStore_host.jar:<other jars...>
```

Appendix

On the implementation on the Sun SPOT and the Host. On the Sun SPOT device, AirStore runs as its own Isolate. This means that multiple Isolates on the same SPOT can access AirStore. On the host, AirStore runs in the same VM as your application. This means other host applications running at the same time will have their own instances of AirStore, and there is the possibility of contention for use of the basestation. To run multiple host apps at the same time, be sure to use the basestation in shared mode. See the “Using the Basestation” section of the Developer's Guide, which is in the doc directory of the sdk.

How AirStore handles late joiners. When AirStore starts up it must determine the current state from some other participant. This is the late joiner problem. Upon start up, AirStore blocks all incoming requests for information (such as a get) and posts a record to AirStore itself, asking to be updated.

Each instance of AirStore runs a thread that is constantly looking for such update requests. When one is found, that thread removes the record, broadcasts out the entire contents of AirStore as a series of puts (which due to the Record Replacement Rule does not change the state of any up-to-date participants). It then replaces the update request record marking it as obsolete. The late joiner is watching for the appearance of its obsolete update request. Upon finding it, the late joiner removes the request, and, knowing that it must have just received a series of puts to bring it up to date, removes all its blocks so its processing can proceed.

When running the AirStoreInspector, you may notice these update request records appear from time to time, typically just after you start a new AirStore application. Such records should quickly disappear, since they are removed by the originator as soon as the update is complete.