Jeff Burch, Agilent

MBARI review feedback

First, I want to thank Tom for inviting me to your review. Your problem domain is very interesting and you have numerous difficult design challenges to overcome for successful ocean deployment. Although oceanography research is very different from the communications systems research that I am involved in, hopefully my comments from my "lessons learned" are helpful to you.

As discussed at the review, my biggest concern is strategies to overcome deficiencies in the wireless network. Because your tests have gone well so far, you may have a better wireless network infrastructure than I have in the 2.5 G cellular networks we have been working with. So, my remarks may be unjustified.

How will the system function when round-trips over the RF link suffer large delays and frequent dropouts?

You know the low bandwidth characteristics of the link and are being careful to limit the amount of data to be transmitted off mooring. Implementing compression technology will be helpful to conserve bandwidth. These are well-understood technologies and should be easy to implement.

With your current use of RMI to access low-level interfaces, I'm concerned that this will result in numerous round-trips as you make many RMI invocations. This may be acceptable in a debug mode but I expect will be painful for normal operations. I suggest you consider a higher-level interface that is able to stream large quantities of desired measurement data off mooring in a single RMI method invocation. For example, input arguments could contain a list of start/stop times for each instrument. Output arguments would contain all the appropriate data from the various message logs. The key issue isn't RMI but minimizing round-trips by using a higher-level streaming API where large chunks of data are retrieved on each trip.

Is the "auto-discovery" scheme for the multi-node mooring going to adversely impact performance?

The proposed multi-node auto-discovery scheme results in the Portal regenerating significant configuration information on each communication cycle. I definitely would characterize how many round-trips over the RF link are required for this to function. If excessive, you could move the discovery function to the mooring host and let it build up the necessary information on all the child nodes. A single round-trip would be used to pull that information back to the Portal.

Another approach would be to have the mooring host generate the information and summarize it via a checksum. I expect that > 99% of the time, the configuration won't change between communication cycles. This checksum would get sent to the Portal on the contact following a sleep / wakeup cycle. The Portal would have cached the details on-shore and could compare checksums. If they differ, a high-level RMI call would be made back to the mooring host to retrieve the whole configuration in one trip. This strategy of using a checksum to represent configuration information has been very effective for us in reducing bandwidth requirements and startup delays.

Is the "wakeup every node every time" scheme for the multi-node mooring fit within your power budget?

Your communication scheme requires that you broadcast from the mooring host to all the child hosts as part of your "is there any data to send" sequence. This requires all the nodes to wake up even if they don't have anything to send to shore.

I really don't know if waking up all the nodes in the multi-node mooring is expensive from an energy point of view. However, I'd measure it and consider turning the problem around like this:

- The "measurement" hosts wake-up when they have useful work to do. This is driven by a local timer that triggers the sampling behavior or in the asynchronous sampling case by an instrument sending serial data
- Each "measurement" host decides when it has useful information to send to shore. This might not be on each measurement cycle. For example, sample every 5 minutes, push data to the mooring host once an hour.
- At some interval, each "measurement" host pushes the "new" portion of the cached data to the mooring host. This wakes-up the mooring host.
- Once new data has been pushed to the mooring host, the measurement host goes back to sleep.
- The mooring host caches the data locally.
    a. If a sufficient amount of data has been cached from all the measurement hosts, it may choose to send the cached data to shore.
    b. If the current time is close enough to the next shore check-in heart beat interval, then the mooring host would also send the data to shore.
    c. Otherwise, the mooring host sets an appropriate timer for the next check-in heart beat and goes back to sleep.
- Assuming no new data from the measurement hosts, the mooring host's timer expires, it wakes up and sends the cached data to shore.
- Note that for the most part, all the nodes except the mooring host wake up ONLY when they have measurements to perform. Otherwise, they stay asleep.
- If the Portal needed to interact with a given measurement host, this would force that host to wakeup via the incoming IP packet.

How do you get the Instrument Puck concepts accepted by this community?

One of the biggest learning's from the IEEE1451.2 standard was the concept of the "transducer electronic datasheet" or TEDS. This is the key enabler for "plug and work". TEDS are also part of the 1451.3 multidrop sensor standard and the 1451.4 dual-mode analog/digital sensor standard.

I like your puck proposal and the partitioning of a SW interface to read and write bytes w/o any additional requirements from the point of view of the instrument. I suggest the following refinements to your basic proposal.

The minimalist implementation would be "read-only" from the host point of view and would be just enough data bytes to provide identity information. You might benefit by studying the 1451.4 "Basic TEDS" format from http://zone.ni.com/devzone/devzoneweb.nsf/Opendoc?openagent&2AB9BE7FAD8379C886256D9B0054573C

The first 64 bits of the transducer TEDS is the Basic TEDS. The Basic TEDS uniquely identifies the transducer and includes the Manufacturer ID (14 bits), model number (15 bits), version letter (5-bit character code), version number (6 bits), and serial number of the device (24 bits). This data is organized according to the format described in Table 1 in a non-volatile memory.

Table 1. Basic TEDS Content

	Bit Length	Allowable Range
Manufacturer ID	14	17 - 16381
Model Number	15	0-32767
Version Letter	5	A-Z (data type Chr5)
Version Number	6	0-63
Serial Number	24	0-16777215

There are differences between the various 1451 standards on how this information is formatted. At this stage, the information is what is important!

So, the "minimalist" puck would only need to provide 8 bytes of memory and support a "read ID" command that would provide these bytes and sufficient information that it was "read-only". Once you get instrument builders to agree on this minimum, you can build from there.

The next most important thing (in my opinion) is to provide more "read-only" storage that describes communication aspects to the instrument. This includes communication features like maximum baud rates, modes (RS232 or RS485), etc. It also would include both a machine-readable and human-readable description of the appropriate commands.

If you could get some agreement on the language to describe these instrument commands, you then can develop a generic instrument driver that has an internal interpreter that uses this information to do the correct thing. In the MBARI case, this generic instrument driver would be subclassed from your Java Instrument class. It would read the necessary information at startup, reconfigure itself, and then be able to operate the instrument.

Note that in this scheme, there IS NO Java driver loaded into the instrument puck. Just a description of what the instrument requires. For example, send this command to the instrument "RD<CR>" and be prepared to receive an ascii string of "comma separated values" followed by a <CR><LF> back from the instrument.

From my experience interfacing to similar RS232 devices, the "generic instrument driver" strategy works very well. Although there are devices that will require new Java code in order to communicate with them (e.g. binary checksums), >80% of RS232 devices usually follow simple ascii protocols with a command → response style syntax.

Next, I'd move onto the calibration problem by having an agreed upon scheme to represent calibration operations and coefficients. This will be most useful if the instrument or the puck provides writeable storage that can be updated by the host. As a minimum, the calibration information can be proprietary to the instrument and all you will be able to do is pass binary or ascii calibration coefficients through your system.

Note that to minimize writeable storage, I'd consider splitting the instrument puck concept into at least two blocks of memory. One block is "read-only" from the host point of view. The other block is writable. Again, make it optional if the instrument provides the writable block.

The next question is how large is the writable chunk? If it is large enough, then the host can put Java byte code there if desired.

Version blues with Java Byte-code

As discussed, having "sufficient information" in the puck so that the host can operate it is the big idea. Plug the instrument into a different host or a different port and you want the system to self-heal.

However, your plans of putting Java byte-code into the puck does have some risks. You may already be up-to-date on this topic, but studying this web site is warranted:
http://java.sun.com/j2se/1.3/docs/guide/extensions/index.html

I'm not up on all the details but I'd try this experiment to see where things break:

1. Build the jar and install in a puck
2. Modify the base class
3. Rebuild the host and try to load the puck

4. Repeat by adding methods, changing method signatures, adding class attributes, etc.

Although I'm not sure this would really solve the versioning problem, you can use the kjc java compiler and store the java source code on the puck. More information on this compiler is located here: http://www.dms.at/kopi/

At least having the source code also on the puck would help with rebuilding it later (even if the jar also contained the byte code).

Miscellaneous

I'm out of time and need to send this to you. Here are some additional comments:

1. Did you consider building GUIs around an embedded http server? We have had great luck with this. Various web pages from the host provide dynamic content. This has been very robust and high performance. This simplifies many of the diagnostic functions to not requiring a custom application on the client.
2. System architects love to debate inheritance versus aggregation. Did you consider aggregation in your design?
3. I have a lot to say on being able to compute on the measured quantities. You will need to invite me back down for this discussion ☺. It definitely seems reasonable that measured quantities would change the behavior. For example, when the temperature from instrument XXX is greater that YYY, force sampling on instrument ZZZ.
4. Be sure to google serialVersionUID to learn about Java version issues with serialization.