Impact of the use of CORBA for Inter-Component Communication in SCA Based Radio

Philip J. Balister
Wireless@VT
432 Durham Hall, Mail Code 350
Blacksburg, VA 24061
Telephone: (540) 231-2958
Fax: (540) 231-2968

Max Robert
Wireless@VT
432 Durham Hall, Mail Code 350
Blacksburg, VA 24061
Telephone: (540) 231-2958
Fax: (540) 231-2968

Jeffrey H. Reed
Wireless@VT
432 Durham Hall, Mail Code 350
Blacksburg, VA 24061
Telephone: (540) 231-2958
Fax: (540) 231-2968

Abstract— **One of the principal problems with SDR in general and SCA in particular is that the overhead incurred through the use of a software infrastructure is difficult to assess. This difficulty stems from the lack of reference point to which to compare this overhead. In an attempt to quantify this overhead, this paper presents the implementation of a simple radio system, in this case an FM receiver, and quantifies the overhead incurred by the use of the SCA, specifically by the use of CORBA within the context of the SCA. Results show that while this overhead is measurable, it is significantly lower than the processing required for the receiver's signal processing task. This shows that, even when the receiver itself is fairly simple, the overhead incurred by the use of CORBA is minor.**

I. Introduction

The Software Communication Architecture [1] (SCA) developed as part of the Joint Tactical Radio System (JTRS) program provides a software framework for a software defined radio (SDR) system. By defining a standard software framework, the JTRS program hopes to improve software reuse across multiple radio systems, reducing the time and cost required to develop radios.

The SCA builds on the industry standards CORBA [2] and XML [3]. CORBA is middleware which provides interfaces between the different software components used by the radio. The primary features of CORBA in the context of SCA-based radios are component location, inter-component communication, and logging. XML is a text-based file format designed to be human readable while maintaining a structure that is easy for a machine to interpret. The SCA defines several XML file formats that contain information describing the radio hardware, softwarem and waveforms that run on the radio.

The work reported on in this paper is based on the OSSIE [4] open source SCA framework developed by Wireless@Virginia Tech. The OSSIE core framework uses two other open source packages, omniOrb which is a vendor implementation of CORBA, and Xerces-c which provides XML parsing. Since the first release of OSSIE in 20004, further releases of the framework have improved SCA compliance and corrected problems discovered while working with the framework. Virgina Tech has delivered several projects based on OSSIE. These projects include a scalability study based on decoding several DRM broadcast signals simultaneously, a cognitive radio demonstration based on Tektronix test equipment, and a narrow band FM receiver.

Additional work involving OSSIE includes a port to the ARM processor portion the TI OMAP processor, a port to the TI C6416 DSP, and a waveform development tool for rapid prototyping of waveforms.

One of the concerns with the SCA is that CORBA uses excessive system resources. This paper examines the impact of CORBA on overall central processing unit (CPU) usage. The results show that CORBA does not require a large amount of processing time to perform the inter-component communication task.

II. Description of Test Waveform

An FM receiver waveform, developed as part of the Chameleonic radio project [5], was used for performance testing. While the receiver was running, profiling software collected run time execution information about the processes that executed and what lines of code were executing. From this data, estimates can be made about what portions of the software executed most frequently.

The receiver is based on a PC running Linux with the waveform developed using the OSSIE core framework. The RF interface is a Universal Software Radio

Peripheral [6], part of the GNU Radio project [7]. The USRP provides an RF front end operating in the 400-500 MHz range, analog to digital conversion, an FPGA that provides a digital down converter, decimation and a USB interface to the PC. An SCA device proxy interfaces the USRP to SCA compatible waveforms. A device proxy provides an interface between the hardware and the software-based components.

There is also an SCA device proxy that interfaces with the sound card in the PC to provide the audio playback function.

The FM receiver waveform consists of three components: a decimator, the FM demodulator, and a receiver controller component. The decimator receives the signal from the USRP device and passes it on to the FM demodulator. The FM demodulator sends the recovered audio to the sound card interface device.

The firmware provided with the USRP may decimate the incoming signal by a factor of 4 to 256. This results in a minimum sample rate delivered to the host computer of 250 ksps. This provides a bandwidth far larger than required for narrowband FM, so the first processing step is to reduce the sample rate by a factor of 10. The incoming samples are low pass filtered, and every tenth sample is output to the FM demodulator.

The FM demodulator receives the complex baseband signal from the decimator component. A phase locked loop (PLL) extracts the audio from this signal. There is an automatic gain control (AGC) prior to the PLL phase detector that maintains the input signal at a level comparable to the signal coming from the numerically controlled oscillator (NCO). This is required to maintain the gain from the phase detector at a constant. After the loop filter there is a DC blocking calculation, which removes any constant level (due to a frequency offset) from the recovered audio. The recovered audio is then sent to the sound playback device.

III. Impact of CORBA on Radio Performance

One question regarding use of the SCA for portable radios is the SCA's dependence on CORBA for inter-component communication. CORBA provides a vendor-independent, platform-neutral structure that allows applications to communicate with each other [2]. These applications do not need to reside in the same processing unit, they may be distributed across processors connected by some form of communication network. Typically, this is a network, but may be a special hardware inter-connect for platforms such as DSPs. There is an extension to CORBA under development by the Object Management Group [8] to provide a structure for specialized transports for use by CORBA; this is called the Extensible Transport Framework (ETF). The transport examined in the report is based on TCP/IP.

Since CORBA contains capabilities beyond what is required for providing inter-component communication for software defined radios, there was some concern that CORBA would require excessive system resources such as processor cycles and memory for the radio built for this project. This section presents information about the impact of CORBA on the final waveform. This report does not look at the latency or data transfer rate limitation, rather it looks at the processor overhead added internally to a component by using CORBA for inter-component communication.

The software performance is measured with a tool called oprofile [9]. Oprofile is a statistical, kernel based profiler that runs on Linux. Since data collection is performed by the kernel, no modifications to the programs of interest are required. A profiler records execution data from the running system that includes instruction pointer and stack pointer information. From this information, a post-processor produces reports showing how frequently the program executes specific functions and particular lines of code. By examining call addresses on the stack, it is possible to divide subroutines time amongst the callers of that function.

An FM receiver was built from the components described earlier in this report. The RF squelch was adjusted so that signal always arrived at the demodulator. After loading and starting the waveform, the profiler was started. This prevented the profiler from counting waveform start-up events. After running the waveform for over thirty minutes, the profiler was stopped and reports were generated.

Three reports were produced for each executable that was profiled. The first report showed overall cpu usage by function, the second showed the call graph and how CPU usage was attributed to different parent functions, and the final report was disassembled source with execution information. Three components were profiled, the USRP interface device, the decimator and the FM demodulator.

Here are the results by routine name for the USRP device:

```
samples  %        symbol name
2180     51.6343  rx_data_process(void*)
626      14.8271  memcpy
53        1.2553  pthread_mutex_lock
50        1.1843  free
```

```
50        1.1843  omni::giopImpl12::
                   marshalRequestHeader
44        1.0422  malloc
41        0.9711  (no symbols)
38        0.9000  .plt
36        0.8527  omniObjRef::_invoke
36        0.8527  (no symbols)
33        0.7816  standardInterfaces_i::
                   complexShort_u::
                   pushPacket
```

Half of the CPU time is spent in the rx_data_process routine. This is the routine that reads data from the USRP and writes it to the decimator via an SCA port. Fourteen percent of the time the component is calling the memcpy functions. Memcpy is part of the standard C library. Using the call graph report, the routines calling memcpy will be identified. In addition to these two routines, the remainder of the CPU time is scattered across many different routines. It is important to note that 25% of the execution is accounted for by routines that use less than 1% of the total time used by the USRP process.

Here is the call graph output for the memcpy function:

```
samples  %          symbol name
  9        1.4377   omni::giopStream::
                     put_octet_array
 20        3.1949   omni::tcpEndpoint::
                     AcceptAndMonitor
597       95.3674   usrp_basic_rx::read
626       14.8271   memcpy
  626      100.000  memcpy [self]
```

The call graph shows the relationship between different interactions with a particular function, in this case memcpy; the functions memcpy calls and the functions that call mempcy. The function of interest, memcpy, is on the line where the function name is not indented. The functions called by memcpy are below it, and the functions that call memcpy are above it. In this case, memcpy does not call any functions, so all the execution time is attributed to memcpy. 96% of the calls to memcpy came from the usrp_basic_rx::read function. This function reads data received over the USB interface from the USRP. These numbers suggest reviewing the code in this routine to look for unnecessary memory copy operations. [1]

After the USRP device collects the data from the USRP hardware, the data is sent to the decimator to reduce the sample to 25 kHz. The decimator component receives data from an SCA port implemented with CORBA, reduces the sampling rate and performs an FIR filter, then sends the data to the FM demodulator via another SCA port implemented with CORBA. This should produce clearer results for CORBA versus component processing. Since the decimator only contains signal processing code and the CORBA code implementing the SCA ports, function usage should be clearer.

Here are the profile results by function name for the decimator component:

```
samples  %        symbol name
44511    71.6659  fir_filter::do_work
14548    23.4233  run_decimation
905       1.4571  complexShort::
                   providesPort::
                   pushPacket
304       0.4895  memcpy
```

For this component 96.5% of the time, the component is executing in one of three routines, the fir_filter (fir_filter::do_work), the main component thread (run_decimation), or the routine that receives data from the USRP (complexShort::providesPort::pushPacket).

From these results we can see that at first glance the CORBA related overhead is very low. We expect to see most of the processing time used by the filter. The 23% used by the run_decimation routine deserves a closer look.

```
14548    23.4233   run_decimation
  43530    74.2035  fir_filter::do_work
  14548    24.7993  run_decimation(void*)
                     [self]
  491       0.8370  standardInterfaces_i::
                     complexShort_u::
                     pushPacket
  69        0.1176  standardInterfaces_i::
                     complexShort_p::
                     getData
```

Inspecting the call graph shows that besides the time allocated to the fir_filter, the remaining time used by run_decimation is in the routine itself. The fir_filter routine shows up since run_decimation calls it with data it receives from the USRP.

Examination of the oprofile report showing interleaved C++ source code lines with the assembly code suggests that much of the time spent in run_decimation is taken up copying the filter output into the CORBA sequences used to send data to the next component. The analysis is complicated by the fact that the compiler optimization process creates assembly code that no longer has a clear relationship with the C++ source. For this case, a simple

[1] In the overall picture, the USRP device does not require a significant amount of processing time, so the relative amount of memcpy calls is not a serious system problem.

routine was written that performs a similar function. The output for this case was compared with the output from the run_decimation routine to verify how assembler sections mapped to C++ source code.

Finally, here is the function profile for the FM Demodulator component:

```
samples  %        symbol name
3156     20.5804  sin
2803     18.2784  cos
1885     12.2921  phase_detect::do_work
1732     11.2944  dc_block::do_work
1675     10.9227  run_demod
1322      8.6208  nco::do_work
1010      6.5862  gain::do_work
193       1.2586  memcpy
92        0.5999  pthread_mutex_lock
81        0.5282  complexShort::
                   providesPort::
                   pushPacket
```

Once again, these routines use just over 90% of the processor time used by the component. The run_demod method uses more time than is expected, but this is again due to copying data into the CORBA sequence used to send the data to the next component.

While the CORBA sequence may appear to create a certain amount of overhead, it is not certain that changing to a different data structure would lead to much improvement. It would be worthwhile to investigate the use of the C++ vector class, or traditional C arrays. However, in the overall picture, the load due to using the CORBA sequence, while measurable, is not out of line with the overall system performance. Furthermore, there are benefits obtained by using the CORBA Sequence. The CORBA Sequence easily integrates with the mechanism used to send and receive data using CORBA, the CORBA sequence provides bounds checking as well as memory management, reducing development mistakes. These features provide good performance in a CORBA environment and help increase radio security by providing a defined error path should a malicious user inject bad data into the radio that causes the software to write data outside the sequence.

IV. Summary

This paper presents the profiling of an SCA waveform, in this case an FM receiver for a push-to-talk system. The presented profiling concentrated on overall CPU usage. Results show that while CORBA's impact on the performance of the system is measurable, they were significanly lower than the signal processing time required for this waveform. This difference is stark, especially when taking into account the relative simplicity of an FM demodulator. Furthermore, when weighed against the benefits associated with the use of CORBA, namely distributed system support, strong typing, and memory management, it is clear that the use of CORBA is compatible with the overall performance needs of an SDR, even one in a constrained environment.

References

[1] "Software Communications Architecture Specification," Final/15 May 2006 Version 2.2.2, Joint Program Executive Office (JPEO) of the Joint Tactical Radio System (JTRS), May 2006. Also available at http://jtrs.spawar.navy.mil/sca/.

[2] "The OMG's CORBA Website." http://www.corba.org/.

[3] "Extensible Markup Language (XML)." http://www.w3.org/XML.

[4] "OSSIE." http://ossie.mprg.org/.

[5] "Chameleonic Radio." http://www.ece.vt.edu/swe/chamrad/.

[6] "GnuRadio:UniversalSoftwareRadioPeripheral." http://comsec.com/wiki?UniversalSoftwareRadioPeripheral.

[7] "GNU Radio - GNU FSF Project." http://www.gnu.org/software/gnuradio/gnuradio.html.

[8] "Object Management Group." http://www.omg.org/.

[9] "OProfile - A System Profiler for Linux (News)." http://oprofile.sourceforge.net/news/.