MIDDLEWARE TRANSPORTS FOR EMBEDDED SOFTWARE RADIO

Roy M. Bell (Raytheon Company, Network Centric Systems, Fort Wayne, Indiana, USA; rmbell@Raytheon.com);

ABSTRACT

Embedded systems are undeniably migrating from hardware to software, and software systems are undeniably using more standard software components. Examples include the use of commercial operating systems and middleware products such as web services and CORBA. Some software developers buck the trend with custom software components in an attempt to gain a short term size or performance advantage, but with the increasing speed of processors, increasing size of memory and increasing demand for more functionality; the long-term trend is to avoid custom components when standardized components meet the need. Using standard components allows developers to reduce time to market or spend their time increasing the functionality and sophistication of their applications.

The JTRS standard supports plug-n-play systems by standardizing the APIs that access and control radio applications and components. These APIs are expressed in both C language and CORBA IDL. Proponents of the CORBA APIs perceive advantages in modularity, reliability, and increased functionality. Proponents of the C language APIs do not extol its virtues, but instead point to CORBA tendencies toward bloated size and performance. These perceived disadvantages are being overcome.

As systems grow larger they become more brittle, take longer to develop, and reduce programmer productivity. Programming teams can regain the advantages of small system development by partitioning systems into independent parts. These parts collaborate to form the complete solution. A partitioning strategy will only be effective if the partitions are highly cohesive, and the inter-communication mechanisms work well.

This paper compares communication mechanisms for embedded systems development. Unlike middleware alternatives such as web services; CORBA is becoming increasingly available on DSPs and FPGAs. CORBA is often a superior strategy, and a good candidate for many radio applications. Its advantages are compelling and its potential disadvantages can be mitigated with customized transports.

1. INTRODUCTION

The value of a system is based on its functionality, its performance, and its perceived reliability. There are plenty of ways to improve the value of a system by spending time, effort, or money. The trick is to find ways to maximize the value obtained for the cost expended. Knuth [1] was the first to extol computer programming as an art form and to say that one can maximize value by mastering the art. Raymond [2] significantly extends this concept by describing the art of UNIX programming. He said that UNIX is designed to be very efficient in launching programs. One should strive to form a system from small, reusable, individualized parts that spring into being when needed, and go away when their job is complete. One should avoid large highly complex, continuously running, do-everything programs filled with lots of threads to process messages. They are much harder to debug. Matloff and Salzman [3] describe the art of debugging, and it begins with the statement that small programs are easier.

Small modules are easier to develop, easier to debug, easier to maintain, and easier to ensure correctness. Large modules require more oversight, a more rigorous development processes, and more levels of management. A large system not only requires more software developers, but also more managers, more planning, more inter-coordination meetings, etc. All of these things grow at a faster rate than the number of lines code. If a program is too big for the available RAM, its performance will be reduced while the OS swaps pieces in and out. Thus partitioning a program into smaller pieces may yield better performance.

A development team might be better off partitioning the implementation into multiple pieces. The sum of the cost for developing the individual pieces could be less and the overall reliability could be higher. Naturally these pieces will have to be combined to form the overall solution. So, the downside to partitioning an implementation is that the individual pieces require runtime coordination and inter-communication.

There are multiple ways to partition an embedded system including 1) separate into programming language functions, 2) use standardized APIs, 3) use Berkeley sockets, and 4) use CORBA. These will all be discussed individually.

2. THE ART OF MODULARITY

Good modularity is often the key to success in software development projects. When partitioning a system; one has to consider the size of each piece and the interfaces between them. If diverse processing elements such as DSPs, FPGAs, and GPPs are available, one also has to consider which piece will operate best on each type of processor. Even though large chunks of software cost more to produce than small ones, it would be a mistake to partition a system into too many pieces. More separate pieces require more runtime coordination and inter-communication. The art of modularity is selecting the right partitioning strategy and finding the right balance between the size of software modules and he need for inter-communication.

Good modularity is also dependent on good interfaces. The success of an interface depends on complexity, throughput, and the degree of separation. A highly complex interface could lead to mistakes, misunderstandings, and rework; and this could defeat the advantage of partitioning. A simple interface that requires high throughput can also reduce the value of the overall system with poor performance. The degree of separation contributes directly to the productivity of the software development team. Moving code to separate functions achieves a low degree of separation. It is an improvement because these pieces can be compiled separately, but the client and implementation are still highly dependent on each other for correct operation. A function not only passes data through parameters and the return value, but a function can also have side effects on global data and it can influence timing, synchronization, and concurrency.

A standardized API does not improve the degree of separation between the client and the implementation. At runtime the client and implementation are still linked together, and it is possible for side effects to occur. However, software development productivity is improved because the behavior and potential side effects are defined by the standard. Highly popular standards such as POSIX are well understood with lots of documentation [1][5][6][7] and many implementations. Productivity is improved precisely because of this dependable behavior.

Maximum separation is achieved through the combination of platform independence, language independence, and location independence. Each of these forms of independence reduce the number of assumptions that can be made about the characteristics of the other parts of the system. Platform independence allows modules to co-exist on different types of processing elements such as FPGAs, DSPs, and GPPs. When a client written in C++ does not know the implementation language of the server; it can operate independently because it cannot make assumptions about things such as the size of an integer, or the way a string is represented in memory. Location independence is perhaps the most important way that client software can be developed independently from the server. When neither the client nor the server knows the location of each other's runtime implementation; they cannot assume that they are operating on the same processing element. Again this is very important to software defined radios, which may be partitioned among a diverse set of processing elements.

3. BERKELEY SOCKETS

The socket API is well documented and well understood [1][8]. Sockets achieve a measure of location independence because they can exchange messages with distributed pieces that reside on the same processor or on different types of processors. A program that relies on sockets is very portable because sockets have a standard API that is available in nearly all operating systems available today. A lot of software does not have to be concerned with timing issues or performance and are perfectly happy with the behavior of sockets. The following list of things could be a problem when building distributed programs using sockets:

- Sockets may not be available if there is a PCI bus (or some other non-Ethernet technology) connecting the source to the destination
- A program will require some type of byte swapping software if the processing element of the destination has a different type of endian architecture
- If there are multiple different types of messages being passed through a socket; there must be a way for the destination to quickly determine which type of message was received, and different message types will likely be forwarded to different destinations
- If a client is written in a different computer language than the destination, there must be an agreement on the way to represent integers, strings, floats, and complex data types, and there will likely be a need to translate from one representation to the other

4. CORBA

CORBA is a well-documented [9][10][11][12] standard defined by the Object Management Group (OMG). It provides the maximum degree of separation through platform independence, language independence, and location independence. Platform independence is achieved through the General Inter-ORB Protocol (GIOP), which defines a standard way to represent things such as integers, floats, strings, composite data structures, constrained and unconstrained sequences, etc. Language independence is achieved through the definition of an Interface Description Language (IDL) and a defined set of mappings to implementation languages such as C, C++, Java, and Ada. Location independence is achieved through the use of an Interoperable Object Reference, which eliminates the need for clients and servers to know the location of each other.

CORBA is usually the best option available for radio software development. It is highly popular, it has a large number of free and commercial implementations, and it offers the possibility to create good modular designs through its achievement of the maximum degree of separation. CORBA implementations are a ready solution for part of the job. Often they do more than is strictly necessary, or more than would be done if a custom solution was implemented. CORBA not only makes it possible to produce a solution in less time, but it also can create a better solution than what was planned.

CORBA is part of the JTRS Software Communication Architecture (SCA), which among other things defines a standardized configuration and deployment mechanisms. The CORBA advantages make it the preferred choice. The only question is whether the perceived CORBA disadvantages will cause another choice to be made.

5. CORBA ALTERNATE TRANSPORTS

Usually CORBA messages are transported through Ethernet and TCP/IP using the Internet Inter-ORB Protocol (IIOP). The combination of TCP/IP and IIOP can be a lot of overhead if the source and destination are on the same processor. It can be especially bad if the source and destination are in the same memory space. Most CORBA implementations provide a short-circuit mechanism for passing messages in the same memory space, but they do not optimize message passing when source and destination are on the same processor, but in different memory spaces. CORBA can introduce other overhead such as mandated exception processing etc. This extra overhead can hurt performance or reduce resources such as processing power or memory space that could be used for other tasks.

The decision to extend CORBA to DSPs or FPGAs will depend on whether there is a CORBA implementation for the chosen device, whether there is room for the implementation of the CORBA infrastructure, and whether the performance is sufficient. The performance of CORBA is entirely dependent on its data transport efficiency. The biggest potential contributor to CORBA inefficiency is the transport technology, which may introduce needless data copying through various levels of the transport implementation.

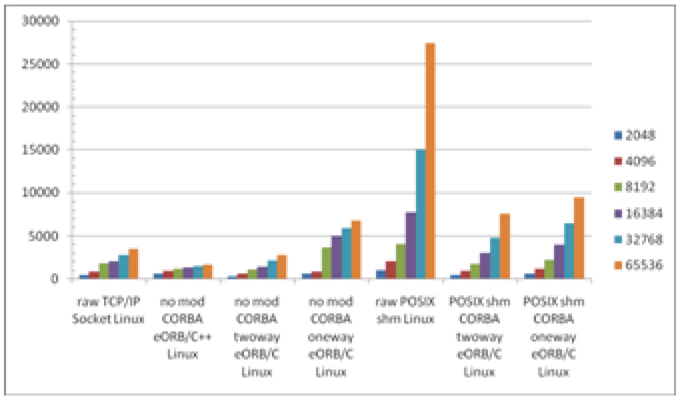

Figure 1: Performance in Megabits/sec for RedHat Linux

Figure 1 shows performance in Megabits/sec for RedHat Linux running in a virtual machine with on an Intel 2.53 GHz dual-core processor with 4 Gbytes of RAM. Each test was run with 2, 4, 8, 16, 32, and 64 Kbyte message sizes. The first 4 tests compare raw sockets to PrismTech e*ORB version 1.6.9. The second and third set of bars show a modest penalty for using CORBA to get the nice modularity benefits of platform independence, language independence, and location independence. In the fourth set of bars we see that eORB/C uses multiple threads to actually beat sockets.

The OMG has adopted the Extensible Transport Framework (ETF) specification, which provides a standard way for users to define a substitute for TCP/IIOP. The fifth set of bars in figure 1 show the performance of shared memory, and the last two set of bars show the CORBA penalty for a shared memory transport.

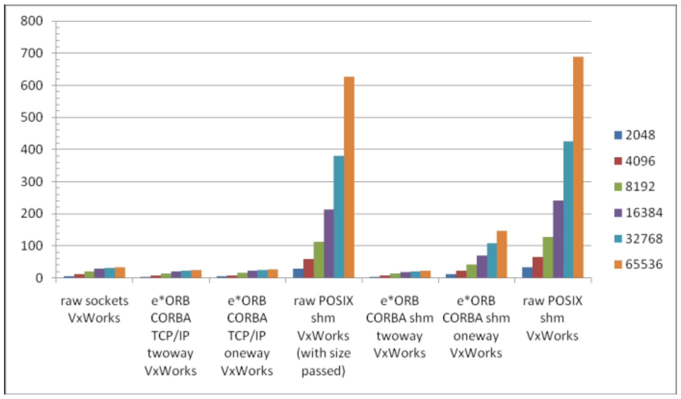

Figure 2: Performance in Megabits/sec for VxWorks

Figure 2 shows the exact same set of tests run on VxWorks 6.8. Both figures show that a CORBA ETF solution built on shared memory provides good performance when compared to TCP/IIOP and only slightly slower than the theoretical maximum achievable with raw shared memory. Clients and servers get all of the benefits provided by CORBA, but without having to depend on a shared memory API or the knowledge that the client and server are running on the same processing element. CORBA ETF can be extended to other onboard transport mechanisms such as POSIX message queues or other off board mechanisms such as transports through a PCI bus.

6. CONCLUSION

Software systems are highly complex. Almost all can benefit from partitioning. The art of modularity is in the choosing of the right size for each partition and the best interfaces between partitions. This is especially true for embedded systems with interfaces to different types of processing elements such as GPPs, DSPs, and FPGAs. Systems that define interfaces using "C" language APIs are less reusable because they mix data transport with the application. CORBA avoids this problem by offering the maximum degree of separation through language independence, platform independence, and location independence. This means that CORBA maximizes modularity and reuse of the application software. One can use a CORBA alternate transport to increase CORBA's availability or to maintain transport performance. CORBA is often the right solution for an embedded software radio implementation.

7. REFERENCES

[1] M. Kerrisk, *The LINUX Programming Interface*, No Starch Press: 2010. ISBN 1-59327-220-0

[2] D. Knuth, *The Art of Computer Programming*, Addison-Wesley: 1998. ISBN 0-201-48541-9

[3] E. Raymond, *The Art of UNIX Programming*, Addison-Wesley: 2003. ISBN 0-13-142901-9

[4] N. Matloff, P. J. Salzman, *The Art of Debugging*, No Starch Press: 2010. ISBN 1-59327-002-X

[5] M. Rochkind, *Advanced UNIX Programming*, Addison-Wesley: 2004. ISBN 013-141154-3

[6] W. R. Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley: 1993. ISBN 0-20156-317-7

[7] D. R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley: 2004. ISBN 0-20163-392-2

[8] W. R. Stevens, B. Fenner, A. M. Rudoff, *Unix Network Programming, Volume 1: The Sockets Networking API (3rd Edition)*, Addison-Wesley: 2003. ISBN 0-13141-155-1

[9] T. J. Mowbray, R. Zahavi, *Essential CORBA*, John Wiley & Sons: 1995. ISBN 0-47110-611-9

[10] A. Vogel, K. Duddy, *Java Programming with CORBA*, John Wiley & Sons: 1998. ISBN 0-47124-765-0

[11] A. Puder, K. Roemer, *MICO is CORBA*, Morgan Kaufmann: 1998. ISBN 3-93258-811-8

[12] T. J. Mowbray, R. C. Malveau, *CORBA Design Principles*, John Wiley & Sons: 1999. ISBN 0-47115-882-8