By Andrew Lyons,
Senior Applications Specialist,
ObjecTime Limited

Bridging the Embedded and Connected Worlds with CORBA

The Internet is a constant reminder that we live in a highly connected world, where people and products communicate quickly and easily. This connected world depends on embedded systems, ranging from massive central office switches and routers to compact cell phones. But connectivity is not just about geographical distribution. Many embedded products are themselves distributed systems, using a back plane or small local network to provide connectivity of processors within the system. Distribution is often used to increase the performance, scalability, and availability of embedded systems. The increasing customer demand for distributed applications, coupled with advances in the enabling technologies of networking hardware and high speed processors, has made distribution a mandatory ingredient of many embedded systems. This article discusses the key issues to consider for a distribution infrastructure, and how Object Request Brokers (ORBs) can extend operating system capabilities to provide a bridge between the embedded and connected worlds. By using commercial solutions, designers can focus on application development rather than infrastructure, thus accelerating product delivery.

DISTRIBUTION INFRASTRUCTURE CHALLENGES

Connected world applications are characterized by the need to distribute software functionality among various processors and provide connectivity among those functional components. Objects are becoming prevalent in embedded software design, and are widely used in distributed systems. If objects are not in the same address space, a distribution infrastructure is required to seamlessly support the same communication mechanisms that objects exploit locally. A distribution infrastructure is middleware that applications use to supplement the underlying capabilities of the Real-Time Operating System (RTOS).

A distribution infrastructure should not only provide seamless object communication to support the initial design of the system but also help designers change the design as the system evolves. It should be optimized for the memory and performance requirements of embedded systems. Thus, there are many key design issues to consider for a project's distribution infrastructure, as outlined in Figure 1. The rest of this section expands on those issues.

Distribution Flexibility
- Location Transparency

Performance
- Middleware
- Transport

Memory
- Small footprint
- Modular feature use

Heterogeneous Systems
- Access Transparency

Project Costs and Risks
- Design & Evolution
- Deployment

Figure 1. Distribution Infrastructure Challenges

Distribution Flexibility

Often the distribution scheme for objects (i.e., the assignment of objects to processors) is heavily influenced by the hardware architecture and predictions of performance impacts. These distribution schemes are often designed early in the project.

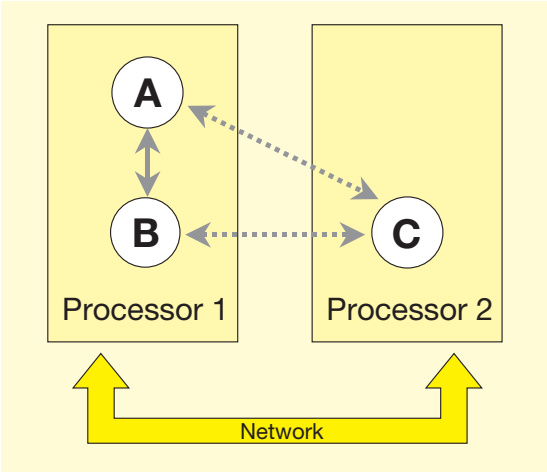

Figure 2. Initial Distribution Scheme

Figure 2 is an example of an initial distribution scheme involving communication among three main objects (A, B, and C), with solid lines indicating local communication and dotted lines indicating remote communication over the network underlying the two processing nodes. At design inception, the developers expect that objects A and B have the highest degree of communication and thus are co-located on the same processor. Object C resides on the other processing node based on an expectation that it incurs high processing demands and involves infrequent communication.

Copyright 1999 by Real-Time Magazine 99-3 (http://www.realtime-info.com)

However, as more experience is gained with that initial distribution scheme, the developers find that there is much higher communication between objects A and C than between objects A and B, and that B has a much higher processing demand than either A or C. In that case, to better match the real requirements, the distribution scheme evolves to that shown in Figure 3. That topology increases the performance of communications between A and C, and better balances the processing loads of the two nodes.

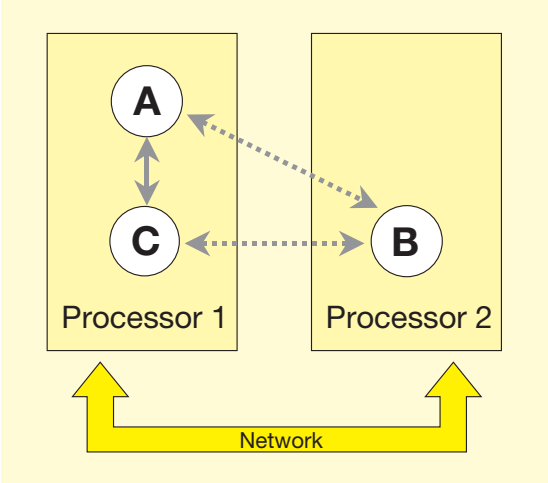

Figure 3. Evolved Distribution Scheme

The preceding example is general, but representative of the design changes that are required as designers learn more about the requirements of the distributed system and the processing and networking performance of their designs. This raises a key requirement of any distribution infrastructure, namely *distribution flexibility*. Developers need the ability to easily change their distribution scheme as they iterate through the design, without the need for significant design rework.

To provide distribution flexibility, ideally the distribution infrastructure should support *location transparency*, where objects should not be concerned with the actual physical location of the target objects they are communicating with (i.e., whether they are in the same process, on the same processor, on another processor on the bus, or on the Internet). When communication is remote, designers should be isolated from the details of finding objects, using the underlying network to communicate with them, and so on.

With location transparency, application design decisions become a separate concern from the physical distribution and network transport decisions. Applications can thus be easily repartitioned across different distribution schemes without major redesign.

Performance

Performance is always a key consideration in embedded systems, and even more so when objects are distributed. Objects need to find other objects quickly, and communicate with them in an efficient, robust, and transparent manner. Thus, the distribution middleware should be designed for high performance.

The underlying choice of transport (protocols and networking hardware) used to move information among nodes of the network also has a significant impact on overall performance. TCP/IP although very ubiquitous and reliable, is not optimal when delay and throughput predictability are important. To address this, the ability to easily accommodate different network transports (through plug-in transports) gives developers the ability to select the right transports for the product requirements, and provides future flexibility, as new transports are available.

Memory

Many embedded systems have memory limitations, so a *small memory footprint* is often essential. Being able to *modularly* select distribution middleware capabilities allows projects to customize minimum configurations matched to their needs. The middleware components themselves should be compact.

Heterogeneous Systems

Support for a *heterogeneous environment* (different languages, operating systems, and processor types) offers both design flexibility and the ability to reuse legacy components or entire systems. Heterogeneity can be supported by the infrastructure providing *access transparency*, where objects are insensitive to the language, data representation, and invocation techniques of the objects with which they are communicating.

Project costs and risks

The impact of the distribution infrastructure on *project costs* and *development risks* is always important. Applications create revenue for the company whereas distribution infrastructure development and maintenance attract costs. Analogous to the decision to use a commercial RTOS rather than building project-specific solutions, the time saved by purchasing a distribution middleware solution rather than building one can help reduce risk in the overall project schedule.

PROPRIETARY SOLUTIONS

The previous section outlined the key challenges of distribution flexibility, performance, memory, heterogeneous system support, and project cost that need to be met by a distribution infrastructure. These problems remain the same regardless of whether a proprietary or commercial solution is selected.

Many embedded projects take a proprietary approach to the problem by building their own distribution infrastructure on top of low level RTOS communication mechanisms such as *sockets*. Sockets support untyped point-to-point information transfer. End points are identified through node addresses and port numbers within the node.

Sockets are a ubiquitous operating system capability, and can provide good performance. However, this is a much lower level of abstraction than the desired ability to make remote object invocation as transparent as invoking a function on a local object. To fill this gap, developers must address many tedious and error-prone issues. The protocol and message formats for the data to be passed through sockets need to be

Copyright 1999 by Real-Time Magazine 99-3 (http://www.realtime-info.com)

defined. Developers must write and maintain their own socket-level programming to determine the node and port addresses of the destination object, initialize the socket to connect to that destination, and subsequently marshal, transmit, and demarshal the data in the appropriate formats. The destination object must demultiplex requests from various objects that are communicating with it. In addition, complex error detection and recovery software must be written. And after all this, the application itself still has to be written.

As socket-based designs are hard-wired for the selected distribution scheme, the resultant lack of location transparency and thus distribution flexibility means that developers can expend significant effort when an application is reconfigured due to inevitable design iterations or technology evolutions.

At a higher level of abstraction than sockets, RTOSs often support various forms of Remote Procedure Call (RPC). RPC compilers can be used to translate a procedural specification into a pair of stubs, for the local and remote sides of the interaction. These stubs communicate with each other by a network, to make a remote procedure call look like a local procedure call. Although much more useful than sockets, there are some disadvantages. RPCs can be used with name services, but sometimes the addressing information is hardwired into the application, thus limiting distribution flexibility. RPCs invoke a specific procedure (with separate data) rather than a specific object with its own specific data. Thus, contrasting with an object-oriented approach, additional state information or parameters must be passed to cause the remote procedure to act on specific elements (e.g., resetting a specific terminal). There is no notion of polymorphism. All procedures with the same name have the same implementation, contrasting with an object-oriented approach where the implementation of the function is class-specific (e.g., "reset" can mean different things depending on the class of object). Only synchronous communication is supported, which can result in unnecessary blocking. Lack of an RPC standard makes it more difficult to support heterogeneous systems due to incompatibilities in RPCs among different vendors.

CORBA – A STANDARD COMMERCIAL SOLUTION

Fortunately, the requirements of a distribution infrastructure are not unique, and the Common Object Request Broker Architecture (CORBA) standard and associated commercial implementations of the standard, have been created to address these needs. The Object Management Group (OMG), an industry consortium of over 800 companies dedicated to accelerating the use of distributed object technology, defines CORBA.

CORBA specifies an Object Request Broker (ORB). An ORB in conjunction with an RTOS provides the desired seamless object connectivity by supporting both location and access transparency. It provides all the underlying mechanisms for connectivity (i.e., object registration, object look up, message delivery, etc.).

CORBA terminology refers to client and server objects. A client object is an object invoking another object through a synchronous or asynchronous message. A server object is one that has previously registered its existence with the ORB and can receive and respond to such client messages. Many embedded objects take on both roles, and thus many CORBA object interactions are really peer to peer.

The remainder of this section describes how CORBA addresses many of the distribution infrastructure issues associated with embedded software development.

Access Transparency and IDL

Access transparency is supported by CORBA via a language-neutral way of defining the interfaces to objects, using an Interface Definition Language (IDL).

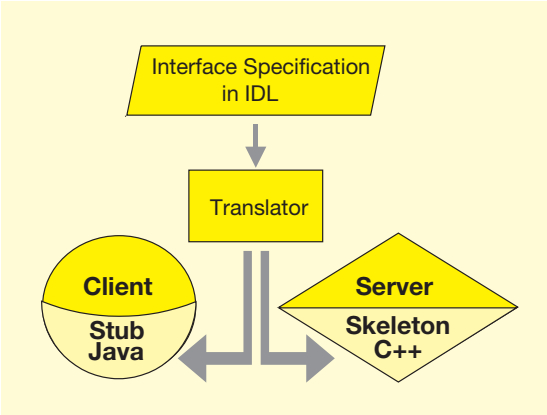

Figure 4. Language Neutral interfaces in IDL

IDL is similar to C++, and can be translated and subsequently compiled to interface classes in the specific languages (i.e., C++, Java, etc) being used for the clients of the object. The interface classes are stubs (proxies or surrogates) that are invoked by the client, that look like local versions of the servers (i.e., a set of function invocations in the client's implementation language). The translator also creates implementation classes in the specific languages of the server objects supporting the interface. The implementation classes on the server side are skeletons into which the developer adds the specific implementation of the operations the interface has defined. Because of the translation into specific languages, there is no run-time penalty for IDL's generic nature.

Location Transparency

Although the implementation of an ORB is complex, how it operates is fairly simple. Each node on the network has an ORB library or core that is linked into the application at build time, as shown in Figure 5. The stubs and skeletons created as part of the IDL translation process isolate the clients and servers from the actual location of the objects with which they are interacting, thus providing location transparency.

For example, to communicate with an object using CORBA, the source object only needs a reference to the target object it wants to communicate with. It does

Copyright 1999 by Real-Time Magazine 99-3 (http://www.realtime-info.com)

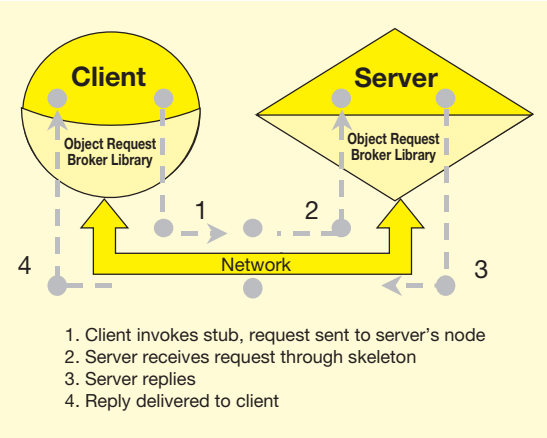

Figure 5. Operation of the ORB

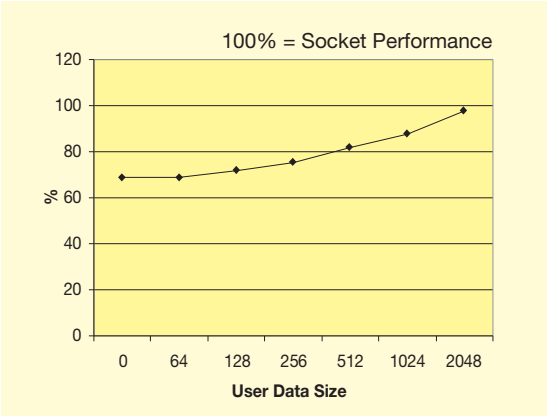

Figure 6. Relative Performance of ORBexpress versus Sockets

not need to know where the object is actually located. The ORB core determines the location of the destination object and passes the parameters to an ORB core on the destination's platform, transparently using the appropriate network transport mechanism. This location transparency in most ORBs is implemented to maximize the performance based on actual location. For example, if the objects communicating with each other are in the same address space (i.e., co-located), there is no parameter marshalling or inter-process communication required. The target object is simply invoked.

CORBA ORBS FOR EMBEDDED SOFTWARE DEVELOPMENT

CORBA prescribes standard ORB interfaces but does not prescribe ORB implementations. Although all ORBs provide location and access transparency, not all ORBs are optimized to address the other key embedded system distribution infrastructure challenges of maximum performance and reduced memory footprint. In addition, run-time royalties often create deployment cost issues for high volume embedded systems.

ObjecTime ORB*express*, a C++ ORB built by Objective Interface based on the CORBA 2.3 standard, was built expressly to address these unique needs of embedded software developers.

ORBexpress performance

ORB*express* integrates efficiently with leading RTOSs such as VxWorks/Tornado. In addition, the internals of ORB*express* were designed with performance as a key criterion. It employs a variety of techniques including quick object look up, compactly-encoded messages to conserve bandwidth and increase messaging throughput, avoiding unnecessary buffer copying, memory allocation/deallocation schemes that reduce memory fragmentation, minimal context switches and system calls, and so on. To measure the resultant performance of an embedded ORB, it is useful to identify the overhead that an ORB incurs over the simplest remote communication mechanism, namely sockets.

Figure 6 shows the relative performance, for various data sizes, of ORB*express* 2.1.1 compared to sockets. The ORB*express* and sockets results were collected in the same environment, an unloaded 10BaseT Ethernet segment between two 200 MHz 2603 PowerPCs running VxWorks 5.3.1-10 under Tornado 1.0.1. The boards had processor-speed caches large enough to contain the benchmarked applications. The underlying transport in both cases was TCP/IP.

This comparison actually puts the ORB at a disadvantage. A fairer comparison (although much more difficult to set up) would be an application where a project has designed proprietary software on top of sockets to create some form of distribution infrastructure. Regardless, the comparison to sockets shows that even at small data sizes there is only about a 30% performance cost in using ORB*express*, and at larger sizes the two approaches are comparable (as the transport itself starts to become the limiting factor).

Transport mechanisms are another key part of the performance picture. TCP/IP provides ubiquity and platform neutrality. Although TCP/IP is a highly reliable transport, this comes at the cost of transmission time not being deterministic. Many embedded systems need a higher performance, more deterministic(1) transport mechanism, or the hardware architecture itself may dictate a different transport. To address this, ORB*express* supports user-definable plug-in transports. With this facility, information can be communicated natively (not TCP/IP emulated) over VME or other busses, proprietary backplanes, FireWire (IEEE 1394), serial lines, ATM, and other media, as easily as the standard support for TCP/IP over Ethernet that ORB*express* provides. ORB*express* can even support multiple transports concurrently. No changes are required to the application when transports are changed, as ORB*express* isolates the application from the transport being used.

ORBexpress memory usage

Minimal memory footprint is important for many embedded systems. ORB*express* allows for modular use of CORBA features. Projects need only load those

(1) For many embedded systems, the timeliness of a message may be more important than how reliably it is delivered. For example, if a control message is received too late to act upon it, it doesn't matter if it was delivered reliably.

Copyright 1999 by Real-Time Magazine 99-3 (http://www.realtime-info.com)

CORBA features they need (for example, the support for dynamic object activation may be excluded). The modular elements of ORB*express* were designed to be compact. For example, a compactly configured ORB*express* for VxWorks on the PowerPC is under 200 KB, and for Windows NT on an X86 it is around 130 KB.

ORBexpress has no run-time royalties

Since ORB*express* was created specifically for embedded systems where per-unit product costs are important, there are no run-time royalties for deploying it in the field.

SUMMARY

Distributed embedded software projects have unique distribution infrastructure challenges. RTOS solutions such as sockets and RPC address many of these issues, but at significant design and maintenance costs. Many projects are using standard commercial ORB solutions such as CORBA to extend the capabilities of their RTOS. CORBA provides distribution flexibility and heterogeneous system support. ObjecTime ORB*express* is a CORBA ORB that addresses the additional key embedded software concerns of performance, memory, and deployment costs. ■

Andrew Lyons has over 17 years of experience in real-time development, ranging from writing kernels for small hard real-time systems to software architectures for large distributed life critical systems. Throughout his development experience, he has applied methods and tools to pragmatic development. Andrew is a senior application specialist at ObjecTime Limited with significant consultancy and training experience. He is part of a collaborative effort with Rational Software on the definition of UML for Real-Time and the Rational Unified Process. He can be reached at andy@objectime.com.

Copyright 1999 by Real-Time Magazine 99-3 (http://www.realtime-info.com)