QNX SOFTWARE SYSTEMS

An Introduction to QNX Transparent Distributed Processing

Yi Zheng, Product Manager, Safety and Security Products
QNX Software Systems Limited
yzheng@qnx.com

Abstract

In a conventional network, devices can share files and data with relative ease. Imagine, however, if any device in a network could, without complex software programming, access the hardware resources—Flash memory, Internet connection, graphics chip, and so on—of any other device as easily as devices now share data. Such is the promise of transparent distributed processing.

In this paper we briefly introduce QNX transparent distributed processing, and explain how this technology can be used to connect disparate interconnected devices into a single logical computer. We present some examples of how transparent distributed processing can be used to meet design requirements ranging from reducing hardware component counts to building fault-tolerant systems that can harness the computing power of hundreds of processors.

What Is Transparent Distributed Processing?

The principle behind transparent distributed processing is simple: to merge all interconnected devices into a single logical computer, where applications can use just one simple programming interface to access both local and remote resources. If an application needs to access, say, a hard disk, it can do so without knowing whether that disk is located on the local device or on another, network-connected device.

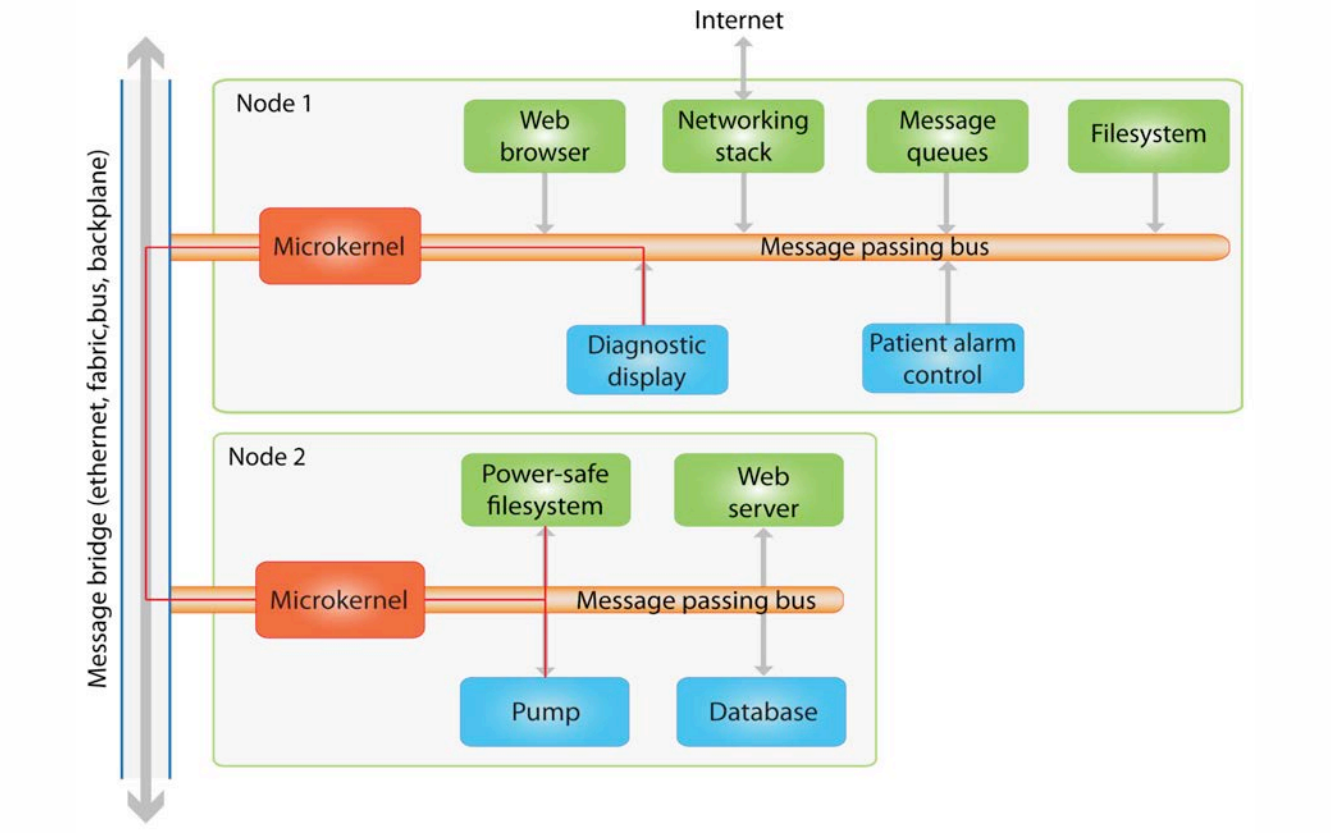

Figure 1. QNX message passing integrates a network of individual devices into a single logical machine. An application running on one node can access the resources of any other node, without special software programming.

Using this capability, we can easily create both low-cost and high-cost variants of a platform. In a car, for instance, we could deploy a lower-cost platform with a single CPU shared by an instrument cluster and an infotainment system, or a higher-cost platform that takes advantage of two separate CPUs. Both the single-and dual-CPU variants would use exactly same software load.

In fact, system designers can leverage the location transparency provided by transparent distributed processing to achieve any number of design goals, from reducing hardware component counts to achieving fault tolerance to building scalable systems that harness the power of hundreds of multicore processors.

Fault-tolerant systems

Besides eliminating unnecessary hardware, the location transparency provided by QNX distributed processing can also simplify the design of fault-tolerant, load-balancing systems. For instance, let's say a machine provides compute services for client applications throughout a network. What happens if that machine fails and a backup machine, serving as a "hot standby," has to take over? With QNX distributed processing, the client applications don't have to be informed that a new machine is now handling their requests, nor do they require special programming to locate the new machine. Any messages they send can be automatically routed to the new destination.

	QNX with Qnet	Linux
Compile time	1. Nothing: shared .h file used for *devctl()* definition.	1. Define function using Interface Definition Language 2. Create/generate Proxy (local end that accepts function call). 3. Create/generate Stub (remote end that executes function call).
Run time (local: call)		
Initialization	1. Open file as per normal.	1. Create and bind socket
Make a call	1. Normal *read()*/*write()*/*devctl()* calls; QNX takes care of proxying message to remote end	1. Call Proxy function 2. Marshall arguments (i.e. expand them into network-consumable form) 3. Send to remote end over socket 4. Block on reply 5. Unmarshall return arguments (i.e. convert from network form to programmatic) 6. Return to caller
Run time (remote: receive)		
Initialization	1. Standard resmgr setup	1. Create and bind socket 2. Create listener
Receive a call	1. Respond as per normal to *read()*/*write()*/*devctl()* calls; QNX takes care of returning results to caller	1. Block on input 2. Unmarshall function arguments 3. Call stub function to execute, passing arguments 4. Marshall return value and return args 5. Send back as response 6. Go back to block on input

Table 1. Comparison of a simple remote procedure call using QNX with Qnet and using another OS, such as Linux, assuming TCP/IP is used.

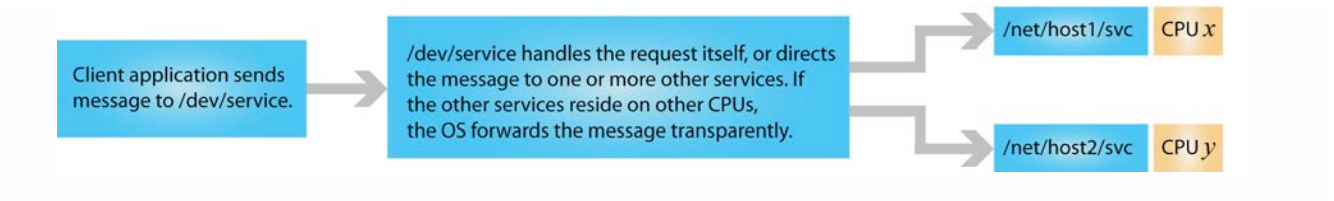

Figure 2. With transparent distributed processing, messages are passed to services indifferently of CPU location.

Distributed message passing

We've discussed what QNX distributed processing can do—but how, exactly, does it work? The answer can be found in the microkernel architecture of the QNX® Neutrino® RTOS, which provides the foundation for this unique networking technology.

In a conventional operating system (OS), most system services—device drivers, file systems, protocol stacks, and so on—run in the OS kernel. Any application that needs to access those services must, as a result, invoke kernel calls. Unfortunately, kernel calls don't cross processor boundaries; they can only invoke services running on the local CPU. It thus becomes difficult, if not impossible, for an application to access system services located on another device.

By implementing a microkernel architecture, the QNX Neutrino RTOS offers a way out of this dilemma. In a microkernel OS, only the most fundamental OS primitives (e.g. threads, mutexes, timers) run in the kernel itself. All other services, including drivers, file systems, and protocol stacks, run outside of the kernel as separate, memory-protected processes.

Since these services don't run in the kernel, they don't have to be accessed by kernel calls. Rather, applications can access the services via message passing, a form of IPC that, when properly implemented, can flow transparently across processor boundaries. An application can, therefore, access virtually any remote service on any node, simply by sending it appropriate messages. In fact, the application can send the exact same messages regardless of whether the service is local or remote.

This message-passing model has another benefit: it forms a virtual "software bus" that allows any service or application to be started or stopped dynamically. Drivers, protocol stacks, and file systems can all be upgraded or restarted, without rebooting the entire system.

Note that Qnet is not the only generic method for making remote function calls. Other methods include CORBA (Unix), which according to some is diffcult to use and hence prone to poor implementation, and ActiveX/DCOM (Windows), which Microsoft proposed as an alternate technology to CORBA, but which it was never able to scale sufficiently well to make generally useful[1].

Industry-standard POSIX interfaces

Message passing provides the foundation on which QNX distributed processing is built. However, there is no need for developers to learn a complex messaging protocol. Unlike with NFS (Network File System), SMB (Server Message Block) or a client/server approach, an application does not need to know anything about the protocols of the nodes where it needs to access services, as long as these nodes are running QNX with Qnet. To exchange messages with remote services, a client application can simply use industry-standard POSIX function calls. Using symbolic links to redirect a resource to the appropriate node further simplifies application development.

Let's examine how this works. In the QNX Neutrino RTOS, any service-providing program (a device driver, for instance) can adopt a portion of the pathname space. Any application can then access that driver by issuing a POSIX *open()* call on the pathname. The application will receive a file descriptor in return, at which point it can begin issuing POSIX file-descriptor calls—such as *read()*, *write()*, and *lseek()*—to access the driver's services. To read some incoming data, for instance, an application would issue a *read()* call, and the underlying C library would transparently convert the call into the appropriate read message.

[1] See, for example, Michi Henning, "The Rise and Fall of CORBA", *acmqueue*, Association for Machine Computing, 1 June 2006. <http://queue.acm.org/detail.cfm?id=1142044>

For instance, Figure 1 shows two processes that need to communicate with each other: in a hypothetical medical device with the HMI, including diagnostic displays and alarm controls, on Node 1 and pump control and monitoring on Node 2. If both processes were running on the same device, the client could invoke the following *open()* call, and the OS would establish a connection between the two processes:

```
/*Open a Flash file system */
fd = open("/dev/fs1",O_RDWR...);
```

In our case, however, the power-safe filesystem is on another device. Consequently, the client could issue the following code instead:

```
/* Open a power-safe file system
   on node 2*/
fd = open("/net/node2/dev/fs1",O_RDWR...);
```

As you see, the code is *identical* to the code used in the single-node scenario, with one exception: the pathname now contains a prefix that specifies the node where the service resides.

Once an application issues a message, one of two things will happen. If the system service receiving the message is on the local processor, the OS microkernel will route the message directly to the service. But if the service resides on another node, then a network manager—an OS process dedicated to forwarding and receiving remote messages—will send the message to that node. A network manager on the second node will then receive the message and forward it to the appropriate process (in this case, the power-safe filesystem). The flowchart in Figure 2 illustrates this sequence of operations.

In addition to using POSIX calls such as *open()*, *read()*, and *write()*, developers can also choose to access the messaging framework directly, using three simple calls: *MsgSend()*, *MsgReceive()*, and *MsgReply()*.

Location transparency though GNS

In the example above, the client application "knows" where the Flash file system is located (Node 2) and uses a pathname to access the service. However, QNX distributed processing also provides a global name service (GNS) that makes the location of system services fully transparent to client applications.

With GNS enabled, an application uses an arbitrary name, rather than a static pathname, to access a service. For example, if an application needed to locate a modem on the network, it could simply specify the name "modem." The GNS server would then locate the modem service on behalf of the application, and the application could then use that service, without knowing where the service was located.

The GNS server is especially useful for networks whose configuration may change at any time. For instance, a system service can be dynamically moved from one node to another, without affecting any of the client applications that communicate with that service.

Redundant links for fault tolerance

To achieve load balancing or fault tolerance, many systems use a cluster architecture that distributes applications and services across multiple networked nodes. These nodes can be several machines connected by a LAN or multiple CPU cards connected to a backplane. If one node fails or becomes inundated with too many requests, other nodes can take over that node's duties until it recovers. Still, the act of distributing applications across multiple nodes can itself create a potential point of failure; namely, the network that connects the nodes together. If any portion of the network fails for any reason, then services can become unavailable. Hence the need for

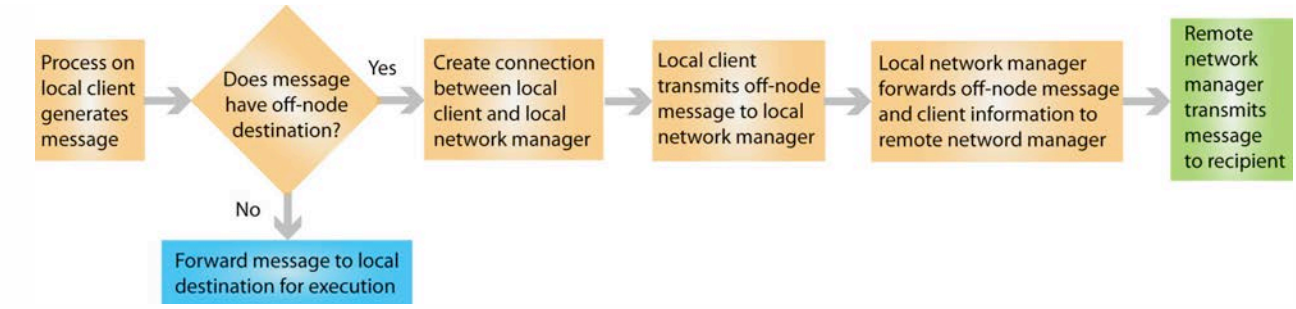

Figure 3. Network managers handle the "dirty work" of forwarding messages to remote nodes.

redundant network links between nodes—and for a software architecture that allows those links to be implemented easily.

To address this need, the network managers that implement QNX distributed processing offer inherent support for multiple links. Moreover, the managers support several Quality of Service (QoS) policies that let system designers control how and when traffic will flow across each link. The QoS policies, illustrated in Figure 4, include:

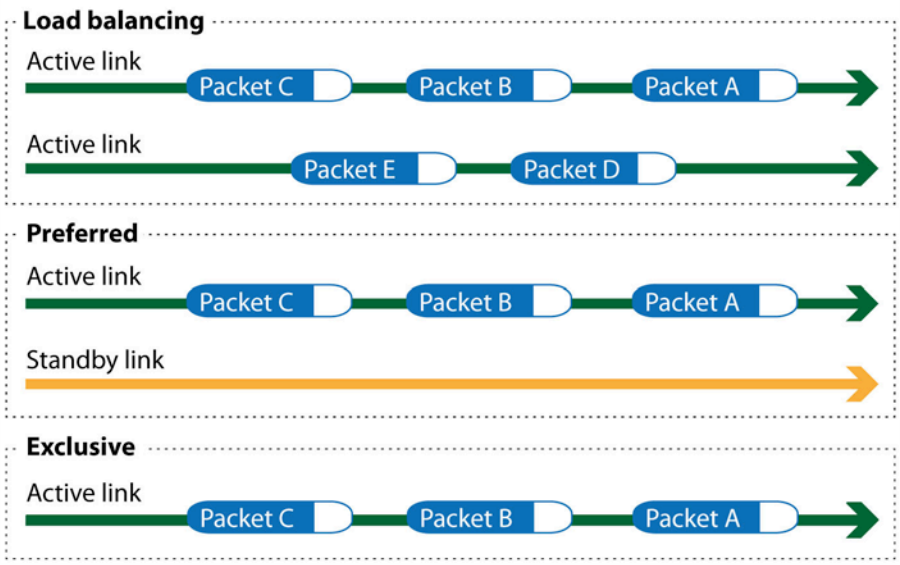

Figure 4. QNX distributed processing offers inherent support for redundant network links, allowing system designers to increase throughput, fault tolerance, or both.

Load-balancing — Queue packets on the link that will deliver them the fastest, based on current load and link capacity. When this policy is in effect, the combined service of all links is used to maximize throughput and allows service to degrade gracefully if any link becomes unavailable. Once a failed link recovers, it can automatically resume sharing the workload.

Preferred — Send out all packets over a specific link until it becomes unavailable, at which point use the second (or third or fourth) link. Qnet will automatically reroute traffic back to the preferred link once it has rejoined the pool of available links.

Exclusive — Only use the specified link. If the link fails, no other link will be used.

Why would you use the exclusive policy? Suppose you have two networks, one much faster than the other, and you have an application that moves large amounts of data. By using the exclusive policy, you can restrict transmissions to only the fast network and avoid swamping the slow network if the fast one fails.

Reliable delivery

To further enhance network reliability, QNX distributed processing supports the following features:

- fragmentation and reassembly
- breakdown of messages into packets within the limits of the packet interface
- assembly of incoming packets into complete messages
- guaranteed delivery of messages and receipt of responses on unreliable packet interfaces

Developers can also incorporate standards-based resolver functionality, including DNS, to route traffic to remote nodes.

Transport independence

By default, QNX distributed processing supports both Ethernet and the Internet Protocol (IP). It can, however, be implemented on any physical link, including LANs, backplanes, system buses, and proprietary switch fabrics.

Conclusion

Today, almost every embedded device must communicate with other embedded devices—or to a larger system— through some form of interconnect, be it a system bus, backplane, switch fabric, LAN, wireless network, or the Internet. This development has, without question, added an extra layer of complexity to embedded design. Unfortunately, most operating systems have not kept pace with this new era of “connectivity.” As a result, developers

must use one programming interface to write applications for a standalone device and use a completely different interface to write applications that will be distributed among two or more networked devices. Because of this approach, applications written for standalone devices can't easily migrate to networked systems, and networked applications can't easily migrate to standalone devices. Software reusability suffers.

With QNX distributed processing, on the other hand, developers can write their applications one way, regardless of how those applications may ultimately be deployed. The same, simple message-passing paradigm that works on a single CPU also works across the network. Code can migrate between standalone and networked systems easily, with little or no modification.

Moreover, any application can, given appropriate permissions, transparently access virtually any resource on any other node, as if that resource were local. System designers can leverage this location transparency to achieve any number of design goals, from reducing hardware component counts to achieving fault tolerance to building massively scalable multi-processor systems. At the same time, QNX distributed processing consumes significantly less overhead than conventional means of interprocessor communication, making it suitable for even resource-constrained embedded devices.

In summary, QNX distributed processing is scalable, efficient, easy to use, and transparent to both the developer and the application. This not only makes it ideal for a variety of embedded designs, but also makes the task of building distributed systems significantly easier than traditional messaging infrastructures.

About QNX Software Systems

QNX Software Systems is the leading global provider of innovative embedded technologies, including middleware, development tools, and operating systems. The component-based architectures of the QNX® Neutrino® RTOS, QNX Momentics® Tool Suite, and QNX Aviage® middleware family together provide the industry's most reliable and scalable framework for building high-performance embedded systems. Global leaders such as Cisco, Daimler, General Electric, Lockheed Martin, and Siemens depend on QNX technology for vehicle telematics and infotainment systems, industrial robotics, network routers, medical instruments, security and defense systems, and other mission- or life-critical applications. The company is headquartered in Ottawa, Canada, and distributes products in over 100 countries worldwide.

www.qnx.com