Communication Mechanisms for Distributed Real-Time Applications

Publish Date: Jan 05, 2012 | 14 Ratings | **4.07** out of 5

Overview

For Aerospace design engineers who use the LabVIEW Real-Time Module for control, simulation, and high-performance data logging and data acquisition applications, distributed shared memory (also known as reflective memory) provides a mechanism for sharing data among the nodes of a distributed system. Unlike ethernet, distributed shared memory provides deterministic operation without added software overhead.

Table of Contents

1. Background and Scope

The objective of this document is to describe distributed shared memory and how it can be used in real-time systems.

Distributed shared memory is a hardware-based communication mechanism for sharing data between computers. Proprietary products are marketed and sold under various names including "Reflective Memory", "Replicated Memory", "Hardware Memory" and "Network Memory" among others. To avoid confusion, this document refers to this technology by the generic name distributed shared memory (DSM).

This paper focuses on typical requirements of 'hard' real-time systems, but it also compares DSM to other data sharing technologies (such as ethernet) that are commonly used in 'soft' real-time applications.

The focus of this paper is on distributed systems that do not share a common backplane. This means multiple distributed computers with one CPU each, as opposed to one computer with multiple CPUs linked through the same backplane.

To learn more about determinism, as well as hard and soft-realtime applications, follow this link to the Real-Time Tutorial.

See Also:
Real-Time tutorial

2. Introduction to Real-Time Distributed Systems

A distributed system can be defined as a system that uses two or more linked computing subsystems to achieve a task. Real-time distributed applications such as simulation, control, or high-performance data logging must be built in a manner that guarantees that all elements of the system operate in a deterministic fashion.

The requirement to share data between subsystems drives the following typical interface requirements:

1. Data availability must be deterministic. The system must guarantee that the data will be available at fixed rates to all the nodes in the system. Since the application is typically time-critical, data determinism is fundamental for correct operation. Distributing data quickly is important, but in this context, getting data “on-time” is more relevant.

2. Data may be shared between all or just some of the nodes in the system. The system should have the capability of selectively sharing data with each and every node in the system. The application software should be able to dynamically select which nodes in the system gets which pieces of data.

3. Low latency communication between nodes. The delay for data to be available at other nodes in the system must be as short as possible. Latency is determined by several factors, including bit rate though the connecting media and overall network traffic.

4. The system must be easily expandable. Scalability is an important factor in distributed systems. The system should be able to seamlessly add or remove nodes to the system with a minimum impact on latency, determinism, and system software. DSM products typically connect subsystem nodes in star or ring configuration. In these architectures, each node usually adds an incremental delay, such that the worst-case latency increases slightly with each added node.

5. Communication between different platforms is desirable. In some cases, a specific computing environment such as Windows or UNIX may be best suited for a particular subsystem function (such as a RAID storage system). In other cases, new technology (such as LabVIEW Real-Time) must be integrated with legacy subsystems (such as VME or VXI). In these situations the ability to share data between multiple platforms is a common requirement in distributed real-time systems.

6. Communication link must be transparent and reliable. The communications mechanism shouldn't add complexity to the application software. The programmer shouldn't have to be concerned with the mechanics of how data is transferred from one node to another. Once the system is configured, the programming interface should require little more than Read and Write operations. In some applications the communication link must also be fault tolerant, which means that the network will still operate even if one of the subsystem nodes fails.

7. Ability to signal and synchronize multiple nodes in the system. Each subsystem needs to be notified when data is available or when the master node requires them to service a specific event. This translates into the need for handshaking and synchronization protocols within the interface hardware.

3. Alternate Approaches

There are several communication mechanisms available for distributed applications. The criteria for selecting a communication mechanism is driven by the application requirements, including the level of determinism (hard vs soft real-time) required by the application.

This section describes the most common alternatives to the DSM approach and describes their pros and cons in relation to hard real-time systems. The focus is on the features that service the needs of distributed hard real-time systems. Keep in mind that these mechanism can still play an important role on sharing non-deterministic data with other parts of the system.

TCP/IP via Ethernet
TCP/IP via ethernet is one of the most common communication protocols used in the computing industry today. It is very reliable and it is included at little or no extra cost.

The main disadvantage of TCP/IP is that it is not deterministic. Some of the characteristics of TCP/IP and ethernet that affect determinism include:

- Network collisions. Message collisions in the media cause the protocol to wait random amounts of time before attempting to retransmitting the data.

- Data acknowledgement. When a sender transmits data, it has to wait until the receiver acknowledges the data. Due to network traffic, CPU activity, and other factors, it is not possible to guarantee how long it will be before the acknowledgement arrives.

Another drawback of the TCP/IP protocol for these applications is that it is not scalable. TCP/IP is a connection based protocol, so it works well when sharing data between two computers (point to point). Adding a third node requires two more point-to-point connections (one for each of the previous two nodes), adding a fourth node requires three additional connections, etc. Since each node must retransmit its data to each of the other nodes, the network traffic can quickly becomes saturated with just a few nodes.

A third drawback is that Ethernet and TCP/IP do not provide a built-in mechanism for synchronization.

UDP
UDP is a networking protocol that presents some advantages over TCP/IP for distributed systems. Since UDP is a connectionless protocol, it doesn't wait for acknowledgement on every datagram sent. This allows for its *read* and *write* calls to be non-blocking, which helps to minimize the jitter in a deterministic loop. UDP also supports multipoint data sharing (multicast) operation.

In spite of these advantages, UDP is not suited for “hard” deterministic distributed systems. Because it is implemented over Ethernet, it also suffers from collisions and random retry timing. In addition, UDP does not guarantee that the data will arrive to the other end, which means that you might lose some data along the way. For most distributed real-time systems, this is an unacceptable characteristic.

DataSocket
DataSocket is a publish/subscribe data sharing protocol created by National Instruments. The concept of publishers and subscribers apply perfectly to distributed applications where there is one data producer and multiple consumers. DataSocket is implemented on top of the TCP/IP protocol, so it has the same fundamental limitations. Since DataSocket is a software protocol, it adds significant overhead and jitter to applications that use it.

Serial Protocols
Serial protocols have been used extensively for communication between computers and hardware devices in general. These protocols include RS-232, RS-422 and many other variations that transmit data sequentially. These protocols are reliable and easy to use.

Serial protocols are not common on distributed real-time applications since they lack synchronization capabilities and the overall transfer rates are considered slow for these type of applications. Some hardware interfaces (e.g. RS-422) support linking multiple systems, but the maximum number of nodes allowed in the system is relatively small.

If transfer rates are relatively slow, utilizing CAN may be a viable solution for sharing serial data within hard real-time distributed systems.

Custom Hardware Interfaces
This approach is sometimes used when the application requires a proprietary protocol or data format that is not supported by industry standards or commercially available products. Hardware protocols can provide determinism and adequate transfer speed, but designing the interface hardware can be challenging, and the cost of development and maintenance may prevent the effort from being cost-effective.

Each of the mechanisms described above provides a medium for data transfer, but it is the responsibility of the programmer to implement the higher-level protocol for synchronizing the data transfer with the application program.

See Also:
CAN Interactive Tutorial

4. Introduction to Distributed Shared Memory

Distributed Shared Memory is an approach to data sharing that is intended for hard real-time systems. In the various DSM product implementations the communication protocol is fully implemented in hardware, which ensures determinism and provides low latency times as well as transparent replication of data between nodes.

Currently there are two DSM products that have driver support for the LabVIEW Real-Time environment:

- the VMIPMC-5565 reflective memory family of boards (http://www.gefanucembedded.com/products/1286) from VMIC
- the SCRAMNet+ Network family of boards. (http://www.cwcelectronicsystems.com/scramnet_sc150_shared_memory.html) from Curtiss-Wright.

Note: To use the VMIPMC-5565 board in a PXI chassis running LabVIEW Real-Time, a third party PMC to cPCI adapter is required.

Each of these vendors provide compatible boards for other platforms including PCI, cPCI, PMC, VME, VXI, etc. Check with each vendor to learn more about their current product offerings.

Note: LabVIEW Real-Time and PXI are open platforms. If you require a DSM board that is not currently supported, it is possible to develop a LabVIEW Real-Time driver for it.

See Also:
Considerations in Implementing LabVIEW Real-Time Drivers
Porting a Windows PCI Device Driver to LabVIEW Real-Time

5. Principle of Operation

In a distributed real-time system, each node hosts a DSM interface which contains a set amount of onboard memory. The memory size depends on the board model and can range from a few kilobytes to several megabytes. Check with each vendor to learn more about their particular offering. After a DSM interface is installed, its onboard memory is available to the host computer. Using a vendor-provided driver, an application can perform read and write operations to the board’s memory.

DSM products usually support connecting up to 256 nodes in the same network. The DSM interfaces are connected via copper or fiber optic cables. Depending on the vendor, a distributed system can be connected as a ring or a star, via a switch. The ring configuration is the most common, since it is easily expandable, but it is not fault tolerant. If one node fails, data sharing fails on the whole network. Star configurations are used in systems where redundancy is needed. If a node fails to operate, the switch changes the configuration so that the remaining nodes in the system can continue to share data.

The following image shows an example of a distributed real-time system in ring configuration.

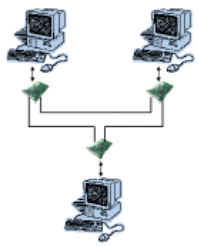

Figure 1. Ring Configuration of a Distributed Real-Time System

When a node writes data to its DSM interface, the hardware automatically replicates the data to the memory on all the other nodes of the network. Each node will be able to see the data almost instantaneously at the same address offset where it was written on the originating node. The transport mechanism, error checking, arbitration, etc is transparent to the user. The driver provides a clean application programming interface (API) that abstracts the high-level operations from these low level details. From the programmer's perspective, the only information needed is the offset and size of the data buffer to read or write.

6. Advanced Features

DSM interfaces typically provide features that enhance data transfer performance and provide hardware synchronization between nodes in the system. This section describes some of these advanced features. For detailed information, please refer to each vendor's product user manual.

Transferring Blocks of Data
In distributed real-time applications, data transfer speed is a very important parameter. Applications typically require sharing blocks of data (e.g. control setpoints and outputs).

In LabVIEW Real-Time, when an application writes (or "posts") data through a DSM interface, the data flow is as follows:

1. Copy the data from the application’s memory to the DSM driver’s memory space.
2. Copy the data from the DSM driver into the VISA driver.
3. VISA then transfers the data into the DSM hardware at a specific offset.
4. The DSM hardware transfers the data to other nodes in the shared network.

If this sequence were to be implemented in the DSM driver software, steps 1 through 3 would require the intervention of the controller's CPU. Byte-by-byte data buffer copying is CPU intensive.

To make this process more efficient, DSM hardware usually incorporates a Direct Memory Access (DMA) engine, which can transfer large buffers of data between application (host) memory and hardware memory without utilizing any CPU time. This approach significantly improves the data transfer rate performance between the host's main memory and the memory on the DSM board, and significantly reduces latency between subsystems.

There are two basic types of DMA operations:

- DMA Write: Transfers data from the host's main memory into DSM interface memory .

- DMA Read: Transfers data from DSM interface memory into the host's main memory.

DSM hardware vendors provide a driver API that abstracts the details of DMA transfers from the user. In general this API is very similar to a typical file I/O API:

1. Configure/Open device for DMA transfer
2. Perform Read or Write operation specifying the data offset and amount of data to read or write.
3. Poll or wait for an interrupt to know that a transfer is complete.

See each DSM LabVIEW Real-Time driver to learn more about their specific API for LabVIEW Real-Time, and the vendor's software reference for the API that they provide for other platforms.

Multiple Node Synchronization
One of the features that makes DSM technology a great fit for distributed applications is the ability to synchronize data generation and consumption between nodes. In typical applications, each node must ‘signal’ the others when new data has been posted or some action needs to be taken by a remote node.

DSM interfaces typically provide this feature via hardware interrupts whose purpose is to inform a host CPU that an event that requires immediate servicing has occurred.

Interrupts may be generated locally or remotely. The DSM can be programmed to generate a local interrupt to inform its host CPU whenever some event (e.g. a particular data location has been updated) has occurred. Similarly, remote interrupts can be generated when a node needs to inform or 'signal' another node that an event has occurred.

By using interrupts, the application developer can create a handshaking mechanism between multiple nodes to effectively synchronize the data transfer. An example execution flow using interrupts in shown below. In this idealized example, node A is streaming data to node B via the shared memory network. Handshaking is needed so that node A can inform node B when new data is ready.

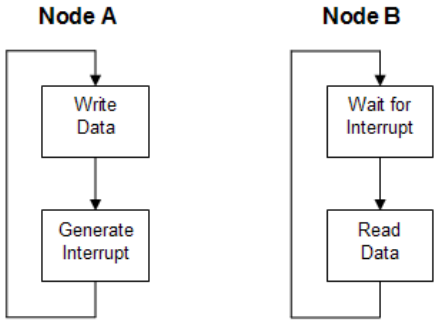

Figure 2. Example Data Streaming Execution Flow

It is important to note that in LabVIEW Real-Time, waiting for an interrupt causes the calling VI to sleep. This alleviates nodes from polling when checking for incoming events or data, allowing the CPU to concentrate on more critical tasks.

Another useful feature is that data can be 'attached' to interrupts, which provides a powerful tool for building a messaging protocol based on interrupts. Using this mechanism, the generating node can attach data indicating the reason for the interrupt. This approach can be used to implement remote commands and other architecture features.

The API for generating and handling interrupts depends on the hardware vendor. The VMIC driver provides a ‘messaging’ scheme where the user specifies the receiving node ID and the message data. When the message arrives at the DSM card, it generates an interrupt to its host CPU. The receiving node uses a ‘wait’ function provided by the driver that causes the calling VI to sleep until a message arrives.

SCRAMNet's interface can be configured to generate an interrupt whenever data is written to a memory location. For example, the user can configure offset '0x2' on the receiving node to generate an interrupt whenever it is updated. When the sending node completes a data transfer, it can write any value to offset '0x2', which will then generate an interrupt on the receiving node. Various memory locations can be configured to generate interrupts, which also supports the creation of a communication protocol between nodes. For example, the receiving node could interpret an interrupt generated by an offset '0x2' update as a ‘data ready’ command and an interrupt coming from offset '0x3' as an ‘abort’ command. The SCRAMNet driver API also provides a 'wait' function that places the calling VI in sleep mode while waiting for an interrupt.

Synchronizing External Hardware
In some applications, it is necessary to trigger other hardware (such as a data acquisition device) when certain events occur on the network. For these use cases, the SCRAMNet boards can be configured to generate a trigger pulse when data in a particular memory offset is updated. This mechanism is very similar to interrupt generation, but sources the pulse on an external connector instead of generating an internal interrupt. The trigger signal can be physically connected to the external device to signal or initiate an operation.

Choosing a DSM Interface
We encourage the use of VMIC, SCRAMNet or any other third party DSM interface hardware. Both VMIC and SCRAMNet provide some unique features that may be better suited for certain applications. The choice of VMIC versus SCRAMNet depends on your familiarity with each product, legacy equipment you may already be using, the memory size you require, transfer rates, or even the interrupt implementation that better suits your needs.

When considering other DSM vendors or interface models, be sure that a driver for the LabVIEW Real-Time environment is available for it, or plan to develop a driver.

See Also:
Deterministic Data Streaming in Distributed Data Acquisition Systems
Deterministic Synchronization of Distributed Simulation Systems

7. Conclusion

Distributed shared memory products provide a flexible and robust mechanism for sharing data and synchronizing nodes in distributed hard real-time applications. Keep in mind that a distributed system has many components, some of which share data but do not require hard determinism. For these components, other communication mechanisms such as UDP or TCP may be best suited. DSM has many advantages for these type of applications but it is also considerably more expensive than other traditional communication mechanisms.