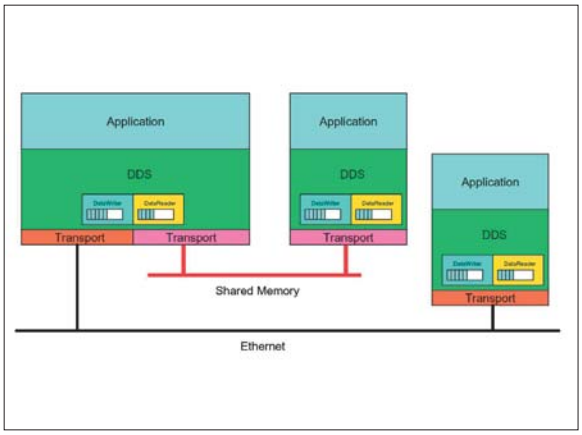
# Designing and debugging real-time distributed systems

**By Geoff Revill,** RTI

*This article identifies the issues of real-time distributed system development and discusses how development platforms and tools have to evolve to address this challenging new environment.*



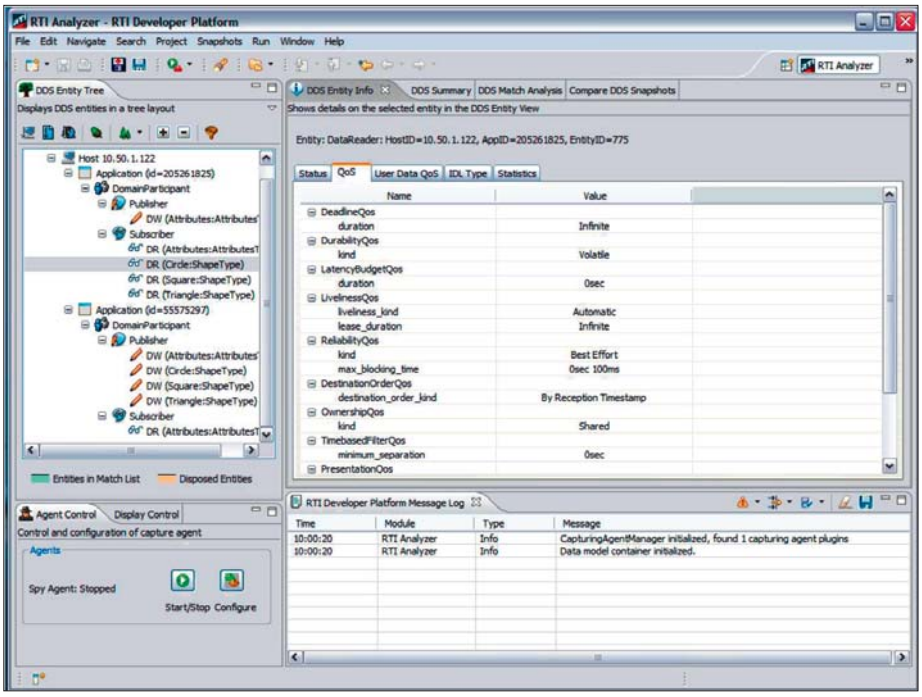

Real-time system designers and embedded software developers are very familiar with the tools and techniques for designing, developing and debugging standalone or loosely coupled embedded systems. UML may be used at the design stage, an IDE during development and debuggers and logic analysers (amongst other tools) at the integration and debug phases. However, as connectivity becomes the norm, systems are becoming ever more distributed; what used to be a few nodes connected together with clear functional separation between the applications on each node, is now becoming tens or hundreds of nodes with logical applications spread across them. Worse, such distributed systems are becoming increasingly heterogeneous in terms of both operating systems and executing processors.

The idea of a "platform" for development has long pervaded the real-time embedded design space as a means to define the application development environment separately from the underlying (and often very complex) real-time hardware, protocol stacks and device drivers. Much as the operating system (OS) evolved to provide the fundamental building block of standalone system development platforms, real-time middleware has evolved to address the distributed systems development challenges of real-time network performance, scalability and heterogeneous processor and operating system support. And as has already happened in the evolution of the standard real-time operating system, new tools are becoming available to support development, debug and maintenance of the target environment – in this case, real-time applications in large distributed systems. From the individual application developer perspective, if the network is now the target system, what are the basic capabilities which must be provided by an application development platform? These are: communication between threads of execution, synchronisation of events, and controlled latency and efficient use of the network resources.

Communication and synchronisation are fairly obvious distributed platform service requirements and are analogous to the services provided by an OS, only now they have to run transparently across a network infrastructure of heterogeneous OS and processors with all that implies in terms of byte ordering and data representation formats. It should ideally use a mechanism that does not require the developer to have an explicit understanding of the location of the intended receiver of a message or synchronising thread so that the network can be treated as a single target system from an application development perspective. There are several middleware solutions which support this approach, such as JMS and DDS (data distributions service) from the Object Management Group (OMG). But only solutions such as DDS explicitly address the third point; controlled latency and efficient use of (target) network resources, which is a critical issue in real-time applications. DDS provides messaging and synchronisation similar to JMS, but additionally incorporates a mechanism called Quality of Service (QoS). QoS brings to the application level the means to explicitly define the level of service (priority, performance, reliability etc) required between the originator of a message or synchronisation request, and the recipient.

DDS treats the target network somewhat like a state machine, recognizing that real-time systems are data driven and it is the arrival, movement, transition and consumption of data that fundamentally defines the operation of a real-time system. Some data is critical and needs to be obtained and processed within controlled/fixed latencies, most especially across the network. Moreover, some data needs to be persisted for defined periods of time so it can be used in computation; other data may need to be reliably delivered but is less time-critical. QoS facilitates all these requirements and more.

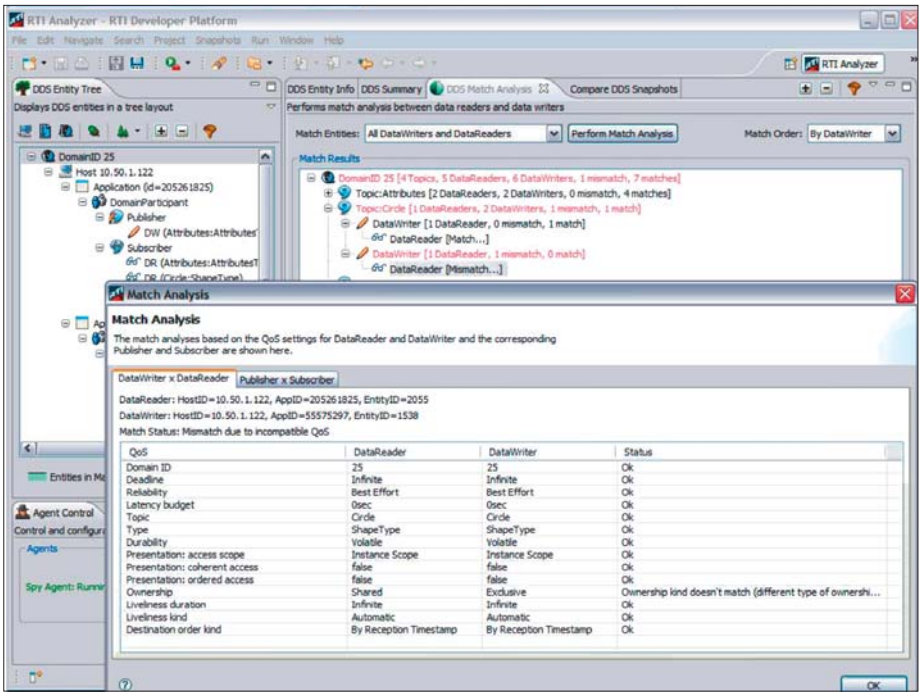
One of the easiest ways to make the networked target environment more debuggable is to de-




*RTI Analyser is a system level debugging tool*

fine strong interfaces between modules that are independently testable. What good middleware does is allow you to completely specify the data interaction through quality of service which forms a "contract" for the application. DDS for example allows a data source to specify not only the data type, but also whether the data is sent with a "send once" or "retry until" semantic, how big a history to store for late-arriving receivers, the priority of this source as compared to others, and the minimum rate at which the data will be sent, as well as many more possibilities. By setting these explicitly many of the soft issues that creep up in integration can be addressed quickly by matching the promised behavior to requested one. DDS middleware will even provide warnings at runtime when contracts are not met.

We do not have a complete application development platform until we have the tools to support the environment. Ask any support or maintenance engineer and they will tell you that they need three things: good documentation, great tools and code written to expose the state and event parameters as easily as possible. Current toolchains that operate on a single node can still be used as normal, and in effect can be used for white box testing in this isolated environment. In addition, the data-driven (state machine) nature of the DDS distributed application development platform means that white box testing can be easily achieved. The real challenge for developers is isolation, identification and correction of the problems that are exhibited at the integration stage, when individual

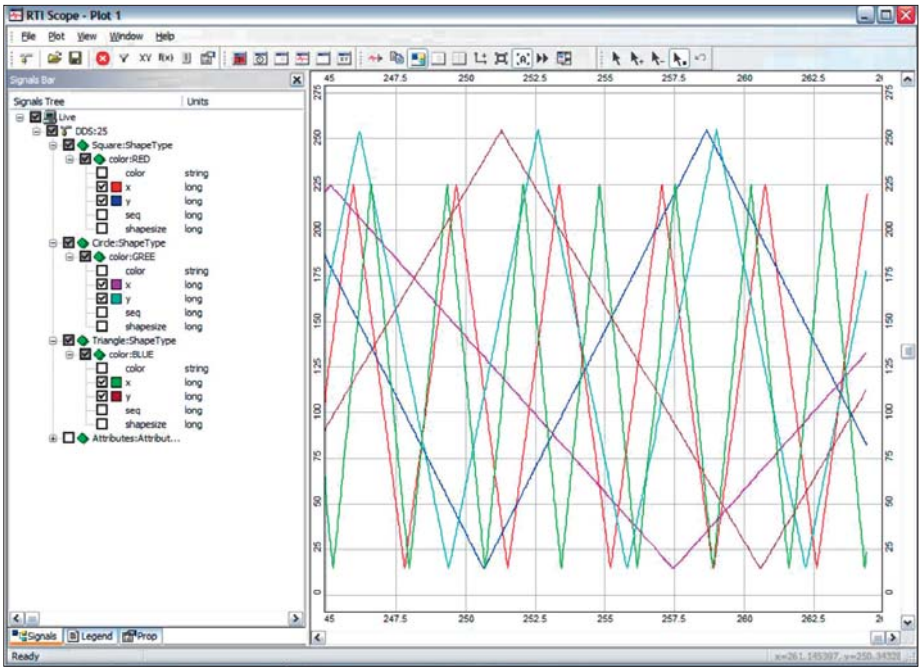



*RTI Analyser showing the QoS mismatch error in "ownership" between a DataReader and DataWriter*

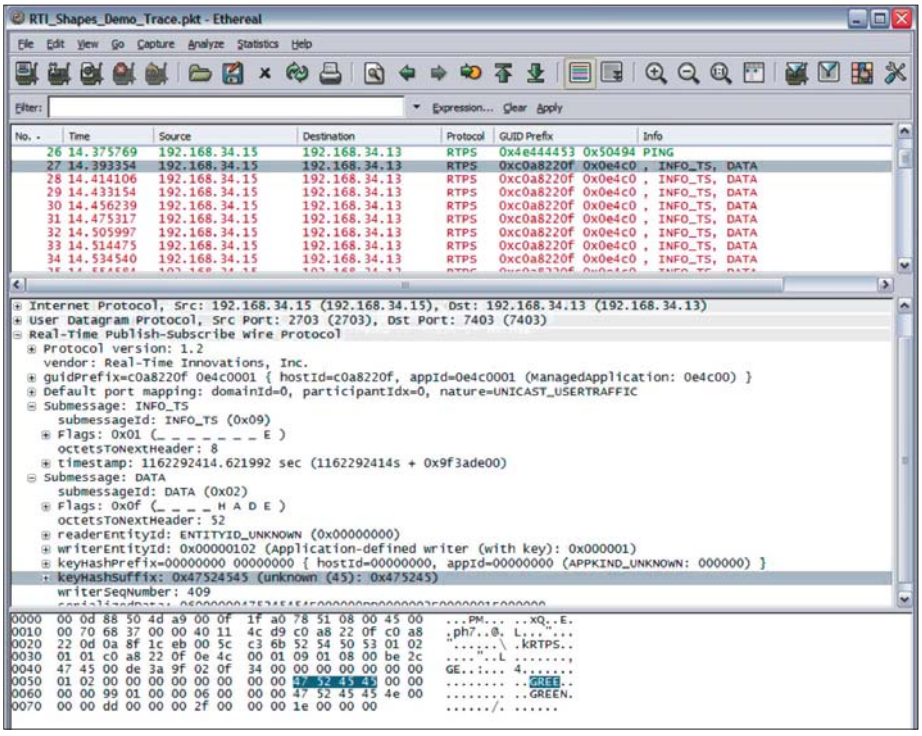
distributed sub-components are connected and the network starts – for the first time - to execute as the target environment.

Most engineers are familiar with debugging within a single-board environment, and will have developed a high degree of debug competence in fixing hard faults, i.e. faults that halt or crash the process. These are relatively easy to debug because you can normally work backwards from the state of the crash or, if you were really lucky, you could get it to crash in a debugger and you were home free. The nastiest hard faults to deal with are normally multi-threading related, so it comes as no surprise that as we move to larger, more complex distributed systems you will see more and more of these types of faults; every node will have its own thread(s) of execution, potentially working on the same data at the same time received from across the distributed system architecture.

Distributed systems are much more likely to be subject to numerous types of soft faults. In these cases, no application crashes, but the warning lights are flashing and the distributed application either performs poorly or not at all. There are numerous types of soft faults, but many of them come down to the synchronisation of data generation and processing across many machines. One example, for instance, is the effect of a single dropped message; if that message is one sample of an update of data it might not be a big deal, but if it is transitional event or command, you could suddenly have the system in an unexpected state. Moreover, you may not be able to detect this until some




*RTI Scope showing DDS topic data plotted against time with an oscilloscope-like display*




*RTI protocol analyser allows you to see the on-wire traffic.*

time after the initial fault occurred, leading to a debugging nightmare.

This is just one type of soft fault. Many others occur regularly: high latencies (either sustained or periodic) which cause control loops to lose stability, massive data dropouts, unexpectedly blocking applications, systems that work in the lab but fail when scaled up, data mismatches between what is provided and what is expected etc. Thus for distributed systems, it is vital to be able to get at the state and event information without stopping or significantly slowing the system.

Starting with the basics: the first thing that you need is a tool that allows you to generate common data types across all your boards and a process that keeps them in synchronisation. If you are using middleware you will normally write your data types in a meta-language (IDL, XML, XDR) and autogenerate the code that handles the data types. Some systems will allow you to create new types on the fly, but beware that this is potentially a source of error since it will be much harder to verify the usage contract on data if the programmer does not know its details. The next tool you need allows you to design the applications and specify the data and QoS requirements. This tool should be used to design as many of the applications as possible so that the QoS contract between senders and receivers is met at design time (much easier than debugging and fixing it later). In an ideal world, this tool should integrate with your normal design methodology. For instance, UML users may wish to consider SparxUML. This tool has interface description components for middleware such as DDS to make it easier to initially set these up. Once your applications are deployed you need to make sure that the communications are happening as intended, QoS parameters are meshed and the system is running. One of the first questions you will need to answer at integration is "are these distributed application functions talking properly?" With the appropriate middleware interrogation tool such as RTI Analyser you can determine that the middleware has hooked up the two applications and you can make sure that the designers of the two application functions actually met specification.

Such a tool also needs to show you which objects are talking, or more importantly, not talking, to each other and if not, suggest why not. You can truly appreciate these tools when you have 3 different subcontractors (or even just free-willed developers) each building part of a distributed application and it comes time to integrate. The root cause of most configuration issues can be found quickly, accurately and with a minimum of debate. You now have great upfront design, good interfaces that people are following and yet it still is not working. This is where distributed system-wide state and event analysis becomes key. Typically there are the following three use cases during the debugging:

Monitoring of overall distributed system health. In this case you might want to see the high-level behavior of most of the applications in the system. Tools such as RTView from SL Corporation allow you to build one or many control panel GUIs or data report views by listening to data put out by the middleware as well as your application. By selectively instrumenting key variables in your application this can be a great first step in isolating system issues and ensuring that your system is running properly. Because tools like RTView leverage the DDS middleware, the location of source information for the displays

does not need to be known. Merely knowing that it exists and in what format it is available (as defined by your data meta-language) and how the data is made available (QoS) facilitates rapid assimilation of the information needed for such useful system overview displays. Typically the applications leveraging this sort of tool will have lots of different data sources, probably at low time resolution, that need to be combined and displayed together to create a meaningful perspective of the system's health. Tools like these are often deployed as part of the maintenance environment for the distributed system and as such include easy-to-use GUI builders that allow end-user-oriented displays of system data and health to be generated.

Getting into the guts of a faulty application. Once you have isolated which nodes are having a problem with the system health tool you may need to get more detailed and higher time resolution data from a few selected applications and their interaction across the network. Tools such as RTI Scope provide this functionality by allowing the user to look at the different data streams into and out of an application graphically, in real-time, without pre-configuration. Think of it as an oscilloscope for the data coming out of an application from anywhere in the network, complete with negative time triggering, multiple plot types (vs time, x vs y), derived signals and the ability to save the data for post processing. RTI Scope still operates at the defined data level, but is designed to capture fewer data sources, in a minimally intrusive manner. It is ideal for capturing data that runs out of bounds, or is delivered outside of its required throughput or performance objectives. Its full knowledge of the underlying middleware implementation means that it can discover the data sources and recipients and connect to them across the network, leveraging the middleware to pull the data through for local analysis and visualisation.

Network Analysis. Sometimes the middleware is attempting to perform the service requested of it by the application, but it is the underlying network implementation itself that is not delivering as expected. Perhaps the router is not routing properly, or there is an address corruption somewhere or any one of a number of other problems. At this point you are left with no choice but to drill down to the wire and see what is happening. You reach for your protocol analyser and it gives you all the UDP or other packet information you need. But it is meaningless unless you can correlate it back up to the application. Well constructed distributed middleware includes a standardised on-the-wire protocol; DDS for example uses the open standard RTPS (real-time publish subscribe), and as you would expect, such a platform includes the ability to monitor the wire traffic and pull out the associated middleware packets, dissecting them for correlation back to the application layer. RTI can help here too with a dedicated protocol analyser, capable of providing a real-time display of all "on-the-wire" activity. ■