The Data-Centric Future, Part II: Performance

A white paper by Stan Schneider

The Rise of Data-Centric Programming

The network is profoundly changing the nature of system design. The "web" is just a first step; the Internet today focuses on connecting people at human interaction speeds. Future networks will connect vast arrays of cooperating machines at rates meaningful to physical processes. These connections make truly distributed applications possible. Distributed applications will drive the future in many areas, from military information systems to financial trading to transportation.

The network itself is only part of the required technology. Today's network makes it easy to connect nodes, but not easy to find and access the information resident in networks of connected nodes. This is changing; we will soon assume the ability to pool information from many distributed sources, and access it at rates meaningful to physical processes. Many label this new capability the "network centric" architecture. We prefer the term "data centric" because the change, fundamentally, is driven by a fast and easy availability of information, not the connectivity of the network itself. Whatever the name, it will drive the development of vast, distributed, information-critical applications.

The importance of performance

Switching from code-centric or architecture-centric thinking to data-centric design is a profound change. Instead of configuring clients and servers, or building data-access objects and invoking remote methods, data-centric design implies that the developer directly controls information exchange. Data-centric developers don't write or specify code. They build a "data dictionary" that defines who needs what data. Then they answer the information questions: How fast is the data coming? Does it need to be reliable? Do I need to store it? Where is it coming from? What happens if a node fails?

Publish-subscribe middleware enables data-centric design. With this approach, nodes simply "subscribe" to the data they need and "publish" information they produce. Information flow is direct, immediate, and efficient.

True power, however, requires the performance to deliver exactly the right data, right now. The middleware can deliver some of this performance by being fast and efficient. Much of the performance, however, comes from efficiently specifying how and when data should be transmitted.

This paper examines the complex issue of the performance of distributed networking software. Part I of this series examined the challenge of defining real-time in a distributed system. It also provided a basic background in real-time systems and middleware architecture and technology, and especially in the publish-subscribe Data Distribution Service for Real-Time Systems (DDS) standard by the Object Management Group (OMG). Now we take a next step, and look at the factors and challenges of delivering the performance required to make data pervasive.

How do we achieve real-time performance?

Pervasive data requires two things: data availability and real-time performance. Real time, fundamentally, means fast enough to reliably effect the environment. Physical processes often operate in environments with millisecond response requirements. In this environment, how do we achieve real-time?

Application requirements---Not!

Logically, discussion of performance should start with a clear definition of the application performance requirements. Unfortunately, in nearly two decades of experience with hundreds of real-time systems, we have (almost) never seen a defensible performance specification, with hard "make it or not" numbers, for a complex system design. That depth of understanding at the start of a project is just not a practical expectation. It becomes somewhat more practical as the project evolves, but it's often only really evident at system integration time.

This may be a shocking claim; don't read it the wrong way. Roughing out the performance expectation is a critical exercise, even if a crisp answer is unlikely. We are simply stating that it is the rare application architect that can state succinctly and authoritatively that the alarm message must propagate from the reporting node to those thirteen other locations within 1.7, and not 1.8, milliseconds.

The best goal, in our experience, is to clearly understand the major performance drivers, and then design a system that can adapt to handle likely loads. This results in the greatest chance of success. It also yields a system that can withstand the inevitable future unexpected demands.

Factors

Even on a single computer, performance is not well defined. To succeed, each single-node part of a distributed application must execute its tasks quickly, reliably, and economically. More importantly, it must complete the computations in time---real time---to successfully interact with its environment.

Connecting many such systems into a network greatly complicates this simple analysis. In a network, issues such as efficient delivery to many simultaneous nodes, efficient utilization of bandwidth, reliability, and recovery from lost transmissions come into play. Complex, distributed real-time systems present the interesting new challenge. Throwing networking into the mix complicates things greatly. Network hardware, software, transport properties, congestion, and configuration effect response time. "On time" may have different meanings for different nodes. Networks merge embedded and enterprise systems, combining the challenges facing both. Even the simplest question of all, "what time is it?" is hard to answer in a distributed system.

But although it may be tough, solving the networked real-time challenge is critical. It is the key to pervasive data. This section overviews the most important factors to consider. Where appropriate, it also outlines the unique approach taken by new technologies like DDS.

Basic network performance metrics

Throughput and latency are the metrics most designers consider. They are, however, often-minor factors in answering the question that matters: will the system work?

Latency

Latency is the time from the moment the sender writes the data until it is received at one or more interested nodes. This is also known as "end-to-end" latency. Since it is difficult to accurately coordinate distributed clocks, latency is usually calculated as half the round-trip time of a message being sent, received, and then returned to the original sender. In some applications, latency "jitter" is also important. Jitter is a measure of how predictable the results are from one write to another.

Latency is a layer game; from the instant the data to send is available, it must pass through the middleware layers, the network stack, the operating system, and the device drivers. Each takes time. Latency also grows as message size increases, making transport time more significant.

For even reasonably large systems, the "1-to-N" latency can be more important than simple point-to-point latency. Most older middleware designs are based on the Transmission Control Protocol (TCP), which implements a single point-to-point reliable connection. Newer middleware can use multicast, the ability to send a single packet to many destinations. DDS can run on multicast networks. Most older technologies, including CORBA, JMS, etc., usually work only via 1-1 unicast. Technologies that can take advantage of multicast can theoretically send to 50 nodes 50 times faster than traditional middleware.

Throughput

Throughput is the total number of bytes sent per unit of time. For small packet sizes, the overhead of passing through all the layers of software dominates throughput. As packets get larger, throughput usually increases; each delivers more data for roughly the same overhead. For very large packets, wire speed can limit throughput.

Most middleware is capable of approaching the common 1Gbit wire speed on today's fast hardware. However, again, the "1-to-N" problem is also important for throughput, and multicast is the key to distributed performance.

Efficiency

How much of the CPU can the middleware use? At design time, this is difficult to answer. Of course, it depends on what else you're doing with the CPU. In our experience, most designs should strive to accomplish their data transport requirements with about 15% of the CPU. That leaves sufficient room for application software and future growth.

Quality of Service

Many designers overlook the effect of Quality of Service (QoS) on performance. By QoS, we mean the semantics of delivery of data, including factors like

- which data to deliver,
- when delivery is expected,
- what to do in case of failure, and
- how many resources to use in the attempt to deliver data.

Middleware designs offer differing control: QoS can be set on a system-wide, per-node, per type of information, or even on a per-data-stream basis. Most networking technologies offer only rough control of global system timeouts. In contrast, DDS, permits the finest control; DDS directly specifies dozens of QoS parameters. Each of these affects delivery performance.

Detailed analysis of QoS effects is beyond this paper. However, three issues are worth discussing: delivery reliability, failure detection and recovery, and deadlines.

Reliability

Unless the network media itself is reliable (rare), networks drop packets. The only way to ensure delivery of lost packets is to retransmit. However, that takes time. Thus, there is a fundamental tradeoff between delivery reliability and time determinism.

Some designs (CORBA, DCOM) are based on TCP and thus offer little to no control over this tradeoff. DDS offers very fine control, all the way down to a per-data-stream level; you can choose reliable or "best efforts" transmission differently for every data topic sent to every node.

Failure detection and recovery

Failure detection and recovery is a complex problem.

The first requirement is to know when part of the system has failed. All middleware designs provide this in some form, either through exceptions on transmit (CORBA, JMS) or explicit “are you alive” protocols (DDS). In general again, DDS provides much finer notification control.

Publish-subscribe systems also offer a first-level failover capability by arbitrating between multiple publishers. The DDS “ownership” concept allows a publisher to own a data stream; if it fails, ownership passes automatically to a backup publisher.

Deadlines

Most networks are “usually” very fast, but sporadically slow or unreliable. Many factors cause this, including operating-system lockouts, stack buffering and overflows, and network-media failure. How does middleware guarantee delivery to a specific node at a specific time in the presence of an unreliable network? The unsatisfying answer: it cannot.

Fortunately, it turns out that most applications can operate without this guarantee. Two things are important: knowing when to expect on-time delivery, and notification in the hopefully rare cases when that expectation is not met. With this information, the application can request the information it needs when it needs it, and react when that request fails.

Scalability

As systems grow, they push the limits of the underlying communication paradigm. The ability to scale is a combination of many factors, ranging from broad system architecture to implementation details like memory allocation. This is a complex problem, made more complex by the many dimensions by which a system can grow. We discuss here only a few of the considerations.

Scalability factors

Nodes

Most think of scalability in terms of physical nodes. Obviously, as this factor grows, more messages are required. However, it is only one of many possible growth factors. Characterizing a system size by the number of nodes is easy and common---but often not that meaningful.

Applications

Each node can support multiple end applications. In fact, most complex systems run several applications, often fairly independent, on each physical node. The number of applications is a better metric than the number of nodes, since the middleware must service each application with the data it needs.

Number of topics

A simple application may have only a few topics to share. Complex applications may require thousands.

Connectedness

A sparsely-connected system requires much less communications traffic than a densely-connected one. More importantly, as a system grows, dense connection topologies can cause exponential growth in the number of required configuration “meta” traffic messages.

Data size

Finally, large databases present performance problems of their own. Accessing this data is a well-studied database problem. Interfacing those databases to high-performance middleware is newly-emerging technology; see below.

Scalability Concerns

Send time

Each time it sends data, the middleware must know what data to send and where to send it. Gathering this information takes time that grows with the system. Sending an update of one topic a second to a few of a dozen applications may be done with simple lists. Choosing which of thousands of applications should receive a complex set of many thousands of topics many times a second may require sophisticated data sorting and lookup techniques.

Discovery time

As systems grow, the problem of which applications need what data grows. This problem is called “discovery”. In a server-based architecture, the server must store and forward all this information. In a peer-to-peer architecture, each node must communicate its needs and capabilities to every other node that shares those needs. As the system grows, the traffic required to manage this information grows. The time required to accommodate that traffic also grows, and can present a limit to scalability.

What does the future hold?

There is a new generation of connected systems in sight. Data-centric architectures will change the world by making information truly pervasive. Pervasive information, available at nearly-instant speeds, will enable much more capable distributed, data-critical applications.

Real-time middleware is the technological key driving the data-centric transformation. Truly pervasive data requires real-time delivery to any location. That requirement imposes many performance and scalability problems.

As we address these challenges, we will develop truly pervasive data. These technologies are evolving rapidly. They will soon deliver the real-time performance, scalability and data integrity required by large distributed embedded systems. Although there are many challenges, a data-centric, pervasive-information future is coming soon.