TWINOAKS COMPUTING INC.

PRACTICAL MIDDLEWARE EXPERTISE

Resource Utilization of Middleware Components in Embedded Systems

Introduction

System memory, CPU, and network resources are critical to the operation and performance of any software system. These system resources impact performance, stability, deployment, and scalability of software systems. Performance will dramatically degrade as memory pages are swapped to disk, applications are forced to wait for CPU time, and network packet loss increases. Stability suffers as process execution becomes less deterministic and applications fail to allocate memory or obtain CPU time. System deployment becomes more complex, when Size, Weight, and Power requirements determine hardware selection and the distribution of software processes. Scalability is impacted as it becomes harder, or impossible to add new features, users, or connections to an existing system that has maxed out its available memory, CPU, or network bandwidth.

A typical approach to software development is to consider the constraints of Size, Weight, and Power (and disk, memory, CPU, and network) during architecture, design, and implementation phases of software development. Software controls can be put in place, and resource usage can be monitored throughout the software development process. Software and designs can evolve as necessary to meet resource constraints.

This process can work well, when a software development team has control of all the software. And in typical embedded applications, this process is the norm. Software projects might make use of COTS hardware, and sometimes a COTS operating system, but all other software components for infrastructure, communications, and the remaining functional requirements are written to meet the constraints of disk, memory, CPU, and network resources.

Gaining insight into resource utilization within your infrastructure components is necessary as embedded systems are developed and deployed, but this can be particularly challenging when using commercial software. How do you gain insight into the resource utilization of a "black box" component? How do you control the memory or CPU utilization? Using the Data Distribution Service (DDS) technology as an example, this paper describes the concepts and tools necessary to determine and analyze the resource utilization of middleware components in embedded systems.

DDS Background

The Data Distribution Service (DDS) is a standardized communication middleware technology, and there are a number of commercial and open source implementations available. Because of it's flexibility, configurability, and performance, DDS is used across a wide variety of industries, including many embedded applications in DoD, space, medical, consumer electronics, energy, and environmental monitoring industries (among others).

A DDS implementation will typically include a library (or set of libraries) that is linked into each application that will

communicate using DDS. For some implementations, a daemon or server process must be running for communications, but this is not required by all implementations.

The DDS technology allows the application to define the data types that will be used for communication at compile time. These data types are defined in a language independent format, and compiled into type-specific DDS code, allowing the software developer to read and write data in a language-natural way. For example, C programmers will use C structures, while Java programmers will use classes.

The DDS technology includes an automatic and dynamic discovery process. Each application using DDS for communications will discover all other applications using DDS, without requiring any configuration of IP addresses or port numbers.

DDS is highly configurable. The standards specify 22 Quality of Service (QoS) policies that are used to configure the behavior of data communications. For example, the reliability, latency, durability, saved history, and organization of data may be configured. These configurations settings can effect resource utilization, some directly, some indirectly.

Many DDS implementations extend this basic set of QoS policies for enhanced control.

System Resources – how are they consumed?

Memory

There are a few aspects to system memory consumption, including static or persistent memory and run-time memory. Static or persistent memory is consumed by executables, libraries, and data files. Run-time memory is consumed by code and static data, the stack, and the heap.

A typical DDS distributed software application will have the following aspects that may consume memory:

1. Executable Code – This category includes application code, libraries, and external daemons or server processes. DDS applications will include type specific generated code, DDS libraries, and may include external daemons or server processes (not all implementations require this).

2. Transport – This category may or may not be appropriate, depending on how the application communicates with its distributed peers. Typical UDP or TCP transports will use buffers to collect data written or received. Many DDS implementations allow the application to select and configure transports. For example, UDP, TCP, shared memory, or serial, all of which have their own configuration aspects.

3. Locally Created Entities – The category is focused on abstracted middleware components. For example, the DDS API requires applications to create entities to read and write data. A publishing

application will create a DDS DomainParticipant (to participate on the DDS network) a DDS DataWriter (to write data of a specific type), and a DDS Topic (to logically write data to). Each of these created entities will consume some amount of memory.

4. Discovery – this category includes memory required for discovery of peer participants. In DDS, each application (or possibly a DDS daemon or server process) will keep bookkeeping information about discovered peer DDS applications and entities.

5. History Caches – This category includes buffers used by the application (or by the middleware) to store sent or received data. DDS contains QoS policies to configure history caches for readers and writers.

CPU

There are many design decisions that will impact the resulting CPU utilization of a software application.

Threading policies is one example. With multi-core CPU's, it is advantageous to parallel process where possible. Communication middleware is an easy place to take advantage of parallel processing: one thread to read data, one thread to manage events (including writing data), and one application thread, is just one way to take advantage of multiple CPU's with one process. However, when deploying on a single-core CPU, especially a low-powered device, threaded applications will require more context switching, in addition to using additional memory. In these environments, less CPU usage can be reduced by deploying a single threaded application.

Another architectural design that impacts CPU utilization is busy waiting versus asynchronous notifications. A communication infrastructure like DDS can offer both design options to system developers.

Network

Data encoding on the wire will impact network utilization: binary formats will generally be more compact than text formats.

Data batching and data aggregation algorithms may reduce network overhead for each message sent. Other algorithms that will impact network usage include the protocol for discovering peer endpoints, and reliability protocol design for handshaking, ack/nacks and re-transmission of missed data packets.

When the distributed software contains one or few writers writing to many readers, the selection of multicast versus unicast will also impact network usage.

Inter-process communications on one machine can be designed to use a local communications mechanism and avoid the network stack. Some DDS implementations provide on-machine transports in addition to network transports. There are significant benefits to being able to use the same communication protocols for all on-machine and off-machine communications, including the flexibility to redistribute software processes across available hardware without significant software re-writes.

Plan early, Monitor often

In any software development process, there is an architecture and design phase, and implementation phase, and a test/integration phase (sometimes multiple phases within each). Changes in design and implementation will happen throughout each of these phases, and in many cases, these changes will be due to resource utilization and limitations. Due to the cost growth of rework in later phases, it is important to plan and monitor resource usage throughout the development process. This is only possible if there are tools to estimate usage (for those architecture and design phases), and measure usage (during the implementation and test phases).

Managing Resource Usage

As software architects and developers, we strive for a balance between conflicting requirements. For example: the balance between performance and memory usage, or the balance between flexibility and code size. With respect to infrastructure software components, it is critical to not only balance between conflicting requirements, but to allow application designers to control which requirements have more importance for any particular application.

A wide range of architectural and design decisions will impact resource utilization, and many of these apply to infrastructure components.

Architecture, distribution, and segregation of software components will impact resource utilization. If we look at a DDS example, segregating software components that do not need to communicate into separate DDS Domains will reduce the amount of discovery memory and network traffic for all DDS applications.

Data architecture will also impact memory and network usage. For example, fixed sized data types allow applications and infrastructure components to pre-allocate space to hold all written and/or received data, but may waste space if data is truly dynamic in size. Unbounded data types (which may include unbounded strings or lists) may use less memory and network bandwidth, but require dynamic memory allocation during application execution.

For example, consider the follow 2 data types (these data types are defined using IDL – a language independent format for specifying data:

```
struct A_fixed {
     String<128> color;
}

struct A_unbounded {
     String color;
}
```

Depending on design constraints and run-time usage, one data type may be preferred over the other.

When using DDS, these data types will be compiled into language specific, type specific code, including the data type declaration. The following may be the resulting (generated) C representation of the data types:

```
struct A_fixed {
    char[128] color;
}

struct A_unbounded {
    char * color;
}
```

Many DDS implementations are threaded, and some allow the application to control the number of threads used, including down to no additional threads.

The DDS standards specify multiple notification options (how the application is notified of events within the infrastructure, including "data is available to be read"): asynchronous (call backs), synchronous (wait on a condition), and polling (calling get_status()).

Applications may improve CPU utilization by aggregating data writes. For example in a threaded DDS implementation, the application may setup a latency budget (QoS policy). With a greater than 0 latency budget, DDS will queue events and reduce the number of times different threads must 'wake up' to perform work. A latency budget may allow the DDS infrastructure to delay (up to the configured budget) the writing of data (for DataWriters) in order to process multiple messages together, and the delivery of data (for DataReaders) in order to notify the application one time for multiple messages.

Another DDS QoS policy allows reading applications to filter received data. Some DDS implementation allow configuration of this filter – to be applied at the writer, before data is written to the network, or at the reader. These options have multiple impacts on resource utilization. Reading applications that perform the filter must 'wake up' for every message written, even if that message will be dropped because of a configured data filter. However, because data is written to all reading applications, the data writer may use multicast to reach many readers with one write on the network. Writing applications that perform the filter must unicast data written to only the reading applications that need to receive it. Depending on the number of reading applications and the filter configuration, this can result in more network traffic than using multicast and filtering at the reader.

Measuring resource usage

There are two ways to measure resource utilization for a software component: from the inside and from the outside. Measuring from the inside requires instrumentation, hooks, or another mechanism where the software component can accurately report is usage of memory or packets written to a network. Inside measuring provides a more accurate measurement of one process or component.

Measuring from the outside is done using external tools, for example: ps, top, tcpdump, or valgrind. Outside measurements are typically a courser measurement. For example, the ps, top, perfmon, and taskmgr tools are not standardized or well documented, and will only measure an entire process. These tools cannot measure the memory or CPU usage

of one library or other component of the application.

External tools are available (for typical desktop environments), and may be used on any software component, even a commercial component. However, they are generally a courser measurement of resource usage.

The most accurate measurements (inside measurements) can only be obtained in COTS products that include hooks and/or instrumentation to allow these measurements.

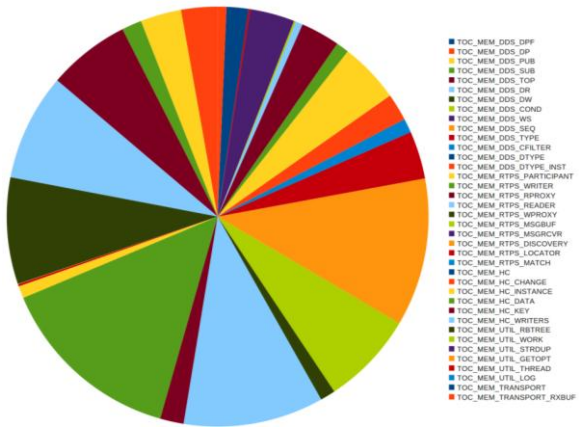

Figure 1: Example of an inside memory measuring tool for DDS

Figure 2: Example of an inside memory measuring tool for DDS

Summary

Resource utilization (memory, CPU, network) is a primary concern when developing deeply embedded components or deploying into a Swap constrained environment. Gaining insight into memory utilization within your infrastructure components is necessary for architects and engineers as they develop and deploy these systems, but this can be particularly challenging when using commercial software. Mismanaged system resources can lead to reduced scalability, poor performance, and increased deployment costs.

It is important to gain insight into the resource requirements of your software components, and especially your infrastructure components. When these components are highly configurable, it is also important to understand the impact of configuration on resource utilization.

The ability to estimate, plan, and monitor resource utilization using inside measurements will ultimately reduce the time required for development and test, as well as reducing overall project risk.

About Twin Oaks Computing

With corporate headquarters located in Castle Rock, Colorado, USA, Twin Oaks Computing is a company dedicated to developing and delivering quality software solutions. We leverage our technical experience and abilities to provide innovative and useful services in the domain of data communications. Founded in 2005, Twin Oaks Computing, Inc delivered the first version of CoreDX DDS in 2008. The next two years saw deliveries to over 100 customers around the world. We continue to provide world class support to these customers while ever expanding.

Contact

Twin Oaks Computing, Inc.
(720) 733-7906
(855) 671-8754
+33 (0)9 62 23 72 20
755 Maleta Lane
Suite 203
Castle Rock, CO. 80108
www.twinoakscomputing.com