Jay Almarode

Configuring server Smalltalk

IN THE PAST, I've described major functional and performance differences between client Smalltalk and server Smalltalk. The client Smalltalk virtual machine operates as a single process that manages objects in virtual memory. Server Smalltalk, on the other hand, operates in a multi-process architecture where the domain of objects can extend beyond the range of virtual memory. Server Smalltalk must coordinate the creation, synchronization, and termination of multiple processes that perform such tasks as: execute a user's Smalltalk code, perform background garbage collection, coordinate multiple users transaction activity, serve disk pages to clients across a network, and manage shared page caches. To provide the needed performance, server Smalltalk is implemented to take advantage of the features of server-class machines, such as shared memory, asynchronous IO, raw disk partitions, and SMP CPU configurations.

Without the ability to to gather statistics, tuning is a shot in the dark.

Obviously, configuring and tuning multi-user, server Smalltalk systems is very different from tuning single-user, client Smalltalk applications. In client Smalltalk applications, the main considerations when tuning are execution speed, runtime memory footprint, and image size. When tuning server Smalltalk systems, additional considerations are system-wide transaction throughput, amount of data transfer to clients, and disk IO rates. The design of the overall system configuration must consider different hardware and operating system parameters, such as the amount of swap space, file system buffers, availability of raw disk partitions, the number of semaphores, or the amount of shared memory.

Due to the multiprocess nature of server Smalltalk, system designers have a number of options when configuring applications to run in a client/server environment. One configuration option is deciding where to execute the behavior of server objects. Each session that is logged into the server has its own virtual machine process to execute server object behavior. Therefore, applications can be configured to create that process on whatever machine it wants. One option is to link the session into the same process as the client virtual machine. This means that both virtual machines are executing within the same virtual memory address space.

Another option is to have the session reside in its own separate process, and communicate with the client Smalltalk through remote procedure calls. This allows a system designer to configure the system so that some session processes reside on client machines, some on the server machine, and others on a third machine. Note that none of these configuration choices impact application code. Application code does not need to know where the execution of server object behavior takes place, and the configuration can be changed with no modifications to application code.

Figures 1—3 illustrate some possible configurations for the location of the session processes. In Figure 1, a single session process is linked with the client process. This makes for fast replication of server objects in the client Smalltalk, but might cause much data transfer over the network when many server objects are read or written. In Figure 2, the session process is separate and resides on the same machine as the client Smalltalk. This configuration enforces data integrity because server objects and client objects reside in different virtual memory address spaces. If a bug in the client application should cause random memory locations to be written with bad data (sometimes called "wild stores"), the server objects are protected by operating system features that prevent one process from writing over another. Depending upon the application, this configuration might suffer from too much data transfer over the network as well.

In Figure 3, multiple session processes reside on the same server machine. One client application has a single session logged into the object server, while the others have multiple sessions logged in. This configuration enforces data integrity also, and data access is faster since the session processes are on the same machine where the disks re-

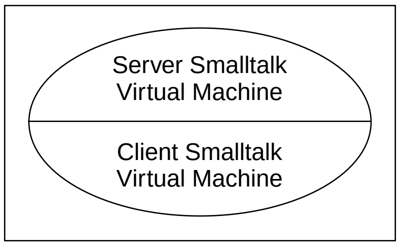

Figure 1. Server and client virtual machines linked.

Using Smalltalk since 1986, Jay Almarode has built CASE tools, interfaces to relational databases, multi-user classes, and query subsystems. He is currently a Senior Software Engineer at GemStone Systems Inc., and can be reached at almarode@gemstone.com.

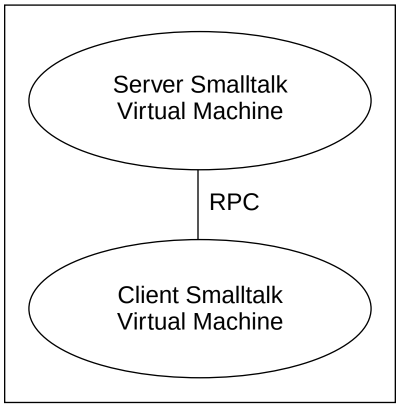
Server Smalltalk Virtual Machine
RPC
Client Smalltalk Virtual Machine

Figure 2. Server and client virtual machines separate.

side. The key advantage of this configuration is that many sessions can take advantage of a shared page cache of server objects. In most applications, a large percentage of objects are read only, and fewer are actually written. In many cases, multiple users are reading the identical objects. Classes and methods are prime examples of objects that are read only during normal application execution. When multiple sessions can share objects in the shared page cache, it saves space and decreases access time, since common objects remain in the cache. A shared page cache can exist on other machines as well; a shared page cache can be created on any machine where a session process is to execute. In general, it is good practice to create a shared page cache on any machine where more than one machine will execute.

There are a number of parameters to tweak when configuring the multiple processes that make up server Smalltalk. The three main kinds of processes in which to configure memory requirements are the server process, the shared page cache and its monitoring process, and the session processes. For each kind of process, various statistics are available to monitor and observe the effects of changing configuration parameters. In GemStone, you can get statistics about various processes that make up the system by executing the expression “SystemcacheStatistics:aProcess Slot”. This message returns an array of information according to the kind of process being described. To get a description of each element in the array, you can execute “System cacheStatisticsDescription”. For statistics gathering purposes, each process is assigned a process slot, and a process executing Smalltalk code can get its own process slot by sending System myCacheProcessSlot. Among the information that you can retrieve for every kind of process is its process name, process ID, session ID, page reads and writes, and cache hits and misses. For the remainder of this column I will discuss the configuration parameters of the three main kinds of processes and describe the relevant cache statistics for tuning performance.

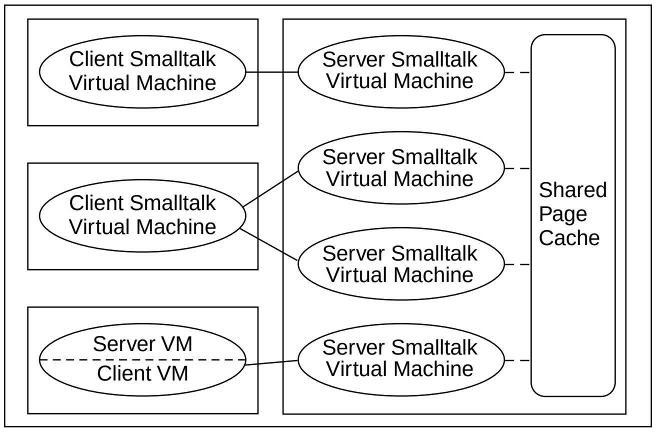
Client Smalltalk Virtual Machine
Server Smalltalk Virtual Machine
Server Smalltalk Virtual Machine
Client Smalltalk Virtual Machine
Shared Page Cache
Server Smalltalk Virtual Machine
Server VM
Client VM
Server Smalltalk Virtual Machine

Figure 3. Server virtual machines sharing pages.

The server process has a number of responsibilities, including synchronizing the transactions of the clients, arbitrating the locking of objects, and allocating object identifiers for clients to use when creating new objects. As the server allocates resources such as transaction records, locks, or object identifiers to each session, it stores this information in its private page cache. The size of this private page cache should be adjusted according to the number of sessions that are typically logged into the server most of the time. If the server’s private page cache is filled up, then it overflows into the shared-page cache, affecting the performance of other sessions.

There are a number of statistics that help measure system throughput. For the server process, one can look at the #TotalCommits statistic to get the total number of transactions committed since the server process started. This can be used to measure systemwide transaction throughput. Another relevant statistic is the #NumberOfCommit Records. This is the number of outstanding transaction records that are currently being maintained by the server process. A large number could mean that there is a long-lived transaction that is preventing the server from reclaiming resources allocated for sessions created later.

The shared-page cache is where multiple sessions share pages of objects. When a session process needs to access an object, it first checks to see if the page containing that object is already in the cache. If so, it reads the page directly from shared memory. If the page is not present, the process reads it from the disk into the cache, where it becomes available to other processes. When configuring the shared-page cache, a system designer considers the total size of the object repository, as well as the number of sessions that will utilize the shared page cache simultaneously.

There are a number of statistics one can look at to monitor the activity of the shared page cache. One statistic that provides some indication of the utilization of the cache is the #NumberOfFreeFrames. This is the number of unused page frames in the shared page cache. Another statistic, #SharedAttached, is the number of pages that are being utilized by more than one process. This indicates the amount of sharing in the cache. When pages in the shared cache are written, they are scheduled to be written to disk when the transaction commits. The statistic #GlobalDirtyPage Count gives the number of pages in the shared cache that have been modified, but are not eligible for writing to disk because they have not been committed yet. If this value is large, then large transactions that write a lot of objects may be taking up space in the cache, or the server’s private page cache may be too small (so it is using the shared page cache for the overflow). This statistic can be compared against #LocalDirtyPageCount, which is the total number of pages that have been modified and are eligible for writing to disk.

As described earlier, each session executes the behavior of server objects with its own *continued on page 28*

decide, 'Why don't we use something less complicated, like C++.'"

The study STIC released last year found that companies adopting Smalltalk were more likely to have followed a formal process in choosing a programming language. "If we can get people to do real comparisons, then Smalltalk has a significant advantage," Phillips concluded. "Smalltalk seems to have to fight its way into an organization, but once it's there, it does pretty well." Smalltalk projects also were twice as likely to achieve their expected goals. "The Smalltalk industry has the opportunity to grow and prosper be cause of the successes that are there. It's a matter of getting the word out," Phillips said.

To Adele Goldberg, the issue is not just teaching Smalltalk, but teaching systems building as opposed to programming. "Too many university computer science curriculums stop at teaching data structures and algorithms," she said. It's not surprising it so hard to recruit people capable of building extensible, adaptable systems. "The most significant part about a system is that once we start it up, there's a maintenance issue. You want it to run indefinitely." And while people can learn the syntax for programming in Smalltalk in an afternoon, "they don't get the systems building part," Goldberg said.

Her solution is LearningWorks, a modified version of the Smalltalk implementations she used to teach programming to 12-year-olds. Its interface is organized into a neat binder of several "books" used for system planning, experimentation, and development, and it feeds students the modern Smalltalk class library a little at a time. Using the internet as a medium for distributing this free tool, she plans to have Open University students collaborate on building LearningWorks systems as class projects.

Students can start by experimenting with rehearsal worlds that illustrate key concepts and provide a context for exercises in organizing behaviors and allocating responsibilities, Goldberg said. Businesses could train their employees by having them create LearningWorks books that represent the essence of the company's framework.

Reg Krock of the Ontario manufacturing firm Maksteel was one of the people who approached Goldberg after her talk to express interest in obtaining a copy of LearningWorks. "One reason is that we have a 67-year-old president of our company. I could give that to him, and he would actually play with it."

Computer systems are the only part of the business that Maksteel's president doesn't fully understand, which makes it harder for him to manage, Krock said. "There's always been a language gap between the CEO and the CIO. What I'd like to do is take some of the mystique out of it."

David Carr is a freelance writer specializing in the object-oriented programming industry. He can be reached at davecarr@pcnet.com.

GETTING REAL

continued from page 22

virtual machine. Each session process has two caches in which to access objects, in addition to the shared page cache. One cache, called the temporary object cache, is where new objects are created. As the execution of server Smalltalk code causes new objects to be created, they are created in a section of memory carved out just for that purpose. This area of memory is garbage collected by generational scavenging techniques, since many newly created objects die early and can be garbage collected soon after their creation. If this cache should become full, then some objects from it must be written to disk, where garbage collection is more expensive. To determine the appropriate size for the temporary object cache, a system designer must consider the total size of all new objects created during a single transaction.

The other cache utilized by the session is the private page cache. This cache is a private area in which to read and write pages of objects. This cache is usually small, since the session primarily uses the shared page cache to read and write objects. If the system is configured not to allocate a shared page cache on the machine where a particular session is executing, then its private page cache size should be increased accordingly.

A session's process can get a variety of information about itself. To monitor garbage collection activity in the temporary object cache, a session can get the #TimeInScavenges statistic to find out the CPU time spent performing in-memory garbage collection, #NumberOfScavenges to find out the number of times the in-memory garbage collector has been executed, or #NumberOfMakeRoomInOldSpace to find out the number of times the oldest generational space filled up (a large number may indicate that the session's temporary object cache size is too small). A session can also find out how well it is using the shared page cache. It can get the #NumberAttached statistic to find out the number of pages that the process is currently using in the shared page cache, and #LocalPageCacheHits and #LocalPageCacheMisses to find out how many times a page was found or not in either the shared page cache or the private page cache. A session can measure its transaction activity by looking at the #NewObjsCommitted statistic to find out the number of newly created objects committed by the most recent transaction, and #NumberOfCommits and #NumberOfFailedCommits to get a cumulative number of successful or failed transactions since the session began.

The statistics described above are but a sampling of the kinds of information to look at when configuring a multiuser Smalltalk system. The key to successfully configuring and tuning such systems is understanding the multiprocess nature of clients and servers, and how different memory spaces and caches are used. Fortunately, tools are available to gather these statistics over long periods of time and then graph the results to analyze overall system performance. Without the ability to gather statistics about each process in system, tuning is a shot in the dark.