IBM'S MAGAZINE FOR AIX DEVELOPERS JULY 1999

AIXpert

FOCUS ON AIX PERFORMANCE TUNING

Optimizing Your System Performance

Inside This Issue:

- AIX Performance Tuning: CPU Usage
- IBM eNetwork Dispatcher and Oracle Net8
- SAS and GPFS

IBM

AIX Performance Tuning: CPU Usage

By Lee Cheng, Wayne Huang, and Matt Accapadi

This article, the first of a series of AIX performance tuning discussions, focuses on CPU usage. It summarizes important information needed to tune the CPU utilization of your AIX operating system, including priorities and scheduling work, standard AIX tools that can be used to determine CPU bottlenecks, how to interpret the reports generated by the tools, and techniques to tune the system.

A review of computing history and operating systems shows that many CPU scheduling policies have been developed by computer scientists. First-in first-out (FIFO), shortest job first, and round robin are just a few. Scheduling policies are important because a single policy may not be best suited to all applications. Some applications in certain workloads can run well in a default scheduling policy. However, the same applications, only with a different workload, may require a scheduling policy adjustment in order to achieve the optimal performance.

Scheduling Algorithms

AIX® implements three different scheduling policies: FIFO, round robin, and a fair round robin.

How scheduling policies are assigned and managed can have a huge impact on system performance; that is, response time and throughput. For example, first-in first-out is a good choice for the job that uses a lot of CPU, but it also can choke out all of the other jobs waiting in line. A plain vanilla round robin gives a "timeslice" or "quantum" to each job in a time-shared manner. As a result, it tends to discriminate against I/O intensive tasks. The fair round robin is "fair" because scheduling priorities change as the jobs accumulate quantums of CPU time during execution. This allows the operating system to demote a CPU hugger so that an I/O bound job will have its fair chance to use the CPU resource.

Two important concepts will be introduced before we discuss the scheduling details: the `nice` value and the AIX priority structure.

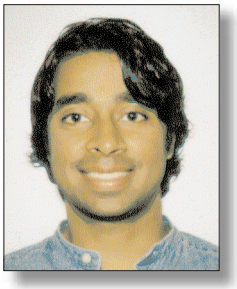
Nice and Renice

AIX has two important scheduling commands: `nice` and `renice`. A user job in AIX carries a base priority level of 40 and a default `nice` value of 20. Together, these two numbers form the priority level.

When starting a job with a `nice` command, say `nice -n 10 myjob`, the number 10 becomes the `delta_NICE`. This number is added to the default 20 to create the new `nice` value 30. In AIX, the higher this number, the lower the priority. Using this example, your job now starts with a priority 70, which is 10 levels lower in priority than the default.

Lee Cheng

Wayne Huang



Matt Accapadi

The `renice` command applies to a job that has already started. A typical `renice` command such as `renice -n 5 -p 2345` will cause process 2345 to have a `nice` value 25. Note that the `renice` value is always applied to a base `nice` of 20, regardless of the process' current `nice` value.

AIX Priority Structure

A thread carries a priority range from 0 to 127; 0 is the highest or the most favorable and 127 is the lowest or least favorable. AIX maintains a run queue in the form of a 128-level priority queue to efficiently support the 128 priority levels of threads.

AIX also implements a 128-bit array to map to the 128 levels of the queue. If a particular queue level is empty, the corresponding bit is set to 0. This design allows the AIX scheduler to quickly identify the first non-empty level and start the first ready-to-run job in that level. See the AIX run queue structure in Figure 1.

First-In First-Out

Although the FIFO policy is the simplest, it is rarely used because of its non-preemptive nature. A thread with this scheduling policy will run all the way to completion unless it gives up the CPU voluntarily by executing a function that would put the thread to sleep—such as `sleep()` or `select()`, get blocked due to resource contention, or have to wait for I/O completion.

The checkout lane at a grocery store is a typical FIFO. Imagine yourself in the checkout lane with only one TV dinner (and you're hungry), but the person in front has a full load in his cart. What can you do? Not much. Since this is a FIFO, you must wait your turn.

Transferring this example to AIX, it is obvious that job response time can suffer severely if several tasks are running in FIFO mode. Consequently, FIFO is rarely used in AIX. Only threads owned by root can be set to FIFO, that is, `SCHED_FIFO`. Also, only root can set a thread to FIFO.

Round Robin

The well-known round robin (RR) scheduling policy is even older than UNIX® itself.

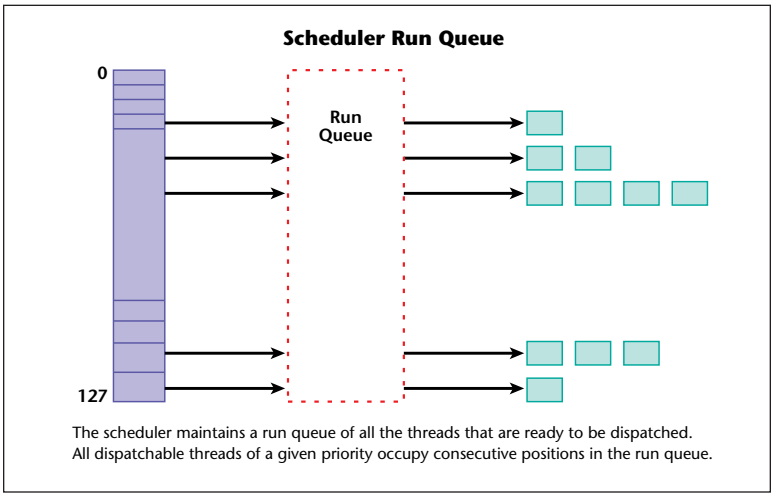


Figure 1. AIX run queue structure

AIX implements RR on top of its multi-level priority queue of 128 levels. At a particular priority level, an RR thread shares the CPU time slices with all other entries of the same priority. A thread is scheduled to run until one of the following occurs: It yields the CPU to other tasks; it is blocked for I/O; or it uses up its time slice. When the time slice is exhausted, the thread is pre-empted and placed at the end of the queue for the next turn to own the processor.

A thread can be preempted because of a higher priority job waking up or a device interrupt (for example, after an I/O is done). For an RR task only, this thread is placed at the beginning of its queue level, because AIX wants to ensure that an RR job has a full time slice before it is moved to the end of the round robin chain. It is important to note that the priority of an RR thread is fixed and it does not change over time. This makes the priority of an RR task persistent (as opposed to the changing priorities in fair round robin) and more predictable.

Since an RR thread has special status, only root can set a thread to run with the RR scheduling policy. To set `SCHED_RR` for a thread, use one of the following application programming interfaces (APIs): `thread_setsched()` or `setpri()`.

SCHED_OTHER

This last scheduling policy is also the default. While trying to establish the fairest policy among tasks, this innovative

SCHED_OTHER algorithm was created with a not-so-innovative POSIX™-defined name.

The AIX SCHED_OTHER is a priority-queue round-robin design at the core, with one major difference: the priority is no longer fixed. If a task is using an excessive amount of CPU time, its priority level should be downgraded to allow other jobs an opportunity to access the CPU.

If a task is at a priority level so low that it does not have an opportunity to run, then its priority should be upgraded to a higher level so it can run. Beginning with AIX 4.3.2, a new concept was also implemented to further enhance the effectiveness of the nice value: If a task is nice (the UNIX nice value) at the beginning, the system will then force it to be nice all the time. We will discuss this new feature later.

Thread Priority Formulas

The priority of a thread can be calculated using the formulas shown in Figure 2. It is a function of the nice value, the CPU usage C, and a tuning factor r.

```
<Formula 1 : for AIX 4.3.1 and Earlier Versions>
Priority = p_nice + (C * r/32)                    (1)

<Formula 2 : for AIX 4.3.2>
Priority = x_nice + (C * r/32 * x_nice_factor)    (2)


Where:
      p_nice = base_PRIORITY + NICE

      base_PRIORITY = 40
      base_NICE = 20
      NICE = base_NICE + delta_NICE

          C is the CPU usage charge.
      The maximum value of C is 120

          If NICE <= 20 then x_nice = p_nice

   If NICE > 20 then
   x_nice = p_nice * 2 - 60  or
   x_nice = p_nice + delta_NICE                   (3)

   x_nice_factor = (x_nice + 4)/64                (4)

       Priority has a maximum value of 126
```

Figure 2. Thread priority formulas

How AIX Calculates the New Priority

The clock timer interrupt occurs every 10ms on each CPU. The timers are staggered so that a CPU's clock timer does not go off at the same time as another CPU's clock timer. When the CPU clock timer interrupt occurs (even before the thread has run for a full 10ms), the thread will have its CPU usage value (that is, the CPU Charge) incremented by one, up to a maximum of 120. If a job does not get a full 10ms slice and it is running with RR policy, the system dispatcher will put this thread at the head of its priority in the run queue to allow it to run again soon.

The recent CPU Charge C is then used to determine the priority penalty and to recalculate the new thread priority. Using formula 1 as a reference (see Figure 2), we know that a newly started user task, which carries a base priority 40, a default nice value 20, with no CPU Charge so far (C=0), will begin with a priority level 60.

Also in formula 1, the value r determines the penalty ratio with a range from 0 to 32. An r value of zero means no CPU Charge penalty, since it makes (C*r/32) to be zero all the time. If r=32, it yields the highest possible penalty for a CPU Charge; each tick (10ms) of CPU usage translates to one priority-level downgrade.

In most cases, the value of r lies near the middle—between 0 and 32. AIX defaults r to 16; that is, every two ticks of CPU Charge become one level of priority penalty. When the r value is high, the impact of a nice value becomes less important since the CPU usage penalty prevails. A smaller r, on the contrary, makes the effect of the nice value more obvious.

Based on this discussion, the effectiveness of the nice value diminishes after awhile because the current CPU Charge grows and gradually becomes the main player in determining the new priority.

This formula has been further modified in AIX 4.3.2 to increase the weight of the nice value in calculating the priority level. This version of AIX introduced two new factors: x_nice and x_nice_factor, that is, "—extra nice" and "—extra nice factor" (see formula 2 in Figure 2).

As you can see from formula 2 and formula 3, the x_nice now has doubled the increased nice value. For the x_nice_factor, it further strengthens the r ratio. For example, an initial nice 16, which gives a nice value 36, will result in a new x_nice_factor of 1.5. That is a 50% higher CPU Charge penalty for the CPU usage part for the lifetime of this thread.

Discount the CPU Usage

It is possible that a thread can get a priority so low that it will never have a chance to run. This would occur if we only have formulas 1 and 2 without a mechanism to push a thread's priority level back up. When a thread runs with SCHED_OTHER, its priority is degraded for its use of CPU time. When it is not running and is waiting for its turn, AIX will try to regain its priority by "decaying" its CPU charges about once a second. The rule is simple: A CPU-bound job should be assigned a lower priority to allow other jobs to run, but it should not be discriminated against the point that it cannot finish itself. All threads' CPU Charge is decayed, based on a predefined factor once per second, as follows:

```
New Charge C = (Old Charge C) * d / 32  (5)
```

A kernel process Swapper does this job. Once every second, Swapper wakes up and handles the CPU Charge decaying for all the threads. The default decay factor is 0.5 or d=16, which "discounts" half of the CPU Charge. With this mechanism, a CPU-intensive job will accumulate CPU Charge, get to a lower priority level, and then be lifted up to a much higher level at the end of a second. An I/O-intensive job, on the other hand, does not vary its priority up and down as much since it generally accumulates less CPU time.

Have You Exhausted Your CPU?

Now that you understand how the AIX scheduler prioritizes the workload, let's look at several commonly used commands. If AIX seems to take too long to finish your workload or it does not respond quickly enough, try these commands to investigate whether your system is CPU-bound: vmstat, iostat, and sar.

The Priority Change History of a Thread

```
Base priority is 40
Default NICE value is 20, assume task was run using the
default nice value

p_nice = base_priority + NICE = 40 + 20 = 60
Assume r = 2 to slow down the penalty increase  (default
r value is 16)
Priority = p_nice + C*r/32 = 60 + C * r / 32

Tick 0    P = 60 + 0 * 2 / 32 = 60
Tick 1    P = 60 + 1 * 2 / 32 = 60
Tick 2    P = 60 + 2 * 2 / 32 = 60
??
Tick 15   P = 60 + 15 * 2 / 32 = 60
Tick 16   P = 60 + 16 * 2 / 32 = 61
Tick 17   P = 60 + 17 * 2 / 32 = 61
??
??
Tick 100  P = 60 + 100 * 2 / 32 = 66

Tick 100  Swapper decays all CPU usage charges for all threads.
          New C CPU Charge = (Current CPU Charge) * d / 32
          Assume d = 16 (the default)
          For the test thread,  new C = 100 * 16 / 32 = 50

Tick 101  P = 60 + 51 * 2 / 32 = 63
```

Figure 3. Change of CPU Charge and the priority of a thread

Fast:

```
main(){for (;;)}
```

Slow:

```
main() {sleep 30;}
```

Figure 4. Priority change of a typical CPU-bound job (fast vs. slow)

We will not discuss all possible ways to use these commands, but instead we will emphasize the information they convey to you. For a detailed description of these commands, see your AIX publication CD-ROM or visit the AIX Web site at *http://www.rs6000.ibm.com/doc_link/en_US/a_doc_lib/aixgen/.*

Common Commands

The `vmstat`, `iostat`, and `sar` commands are used frequently for CPU monitoring. You should be familiar with the usage and the meaning of the generated reports.

vmstat

The `vmstat` command provides an overview of resource utilization through a report of CPU, disk, and memory activity in a one-line-per-report format. The sample output in Figure 5 is generated on an AIX 4.3.2 system by running `vmstat 3 5`.

This report was generated every three seconds as requested. Since a count of five was specified following the interval, reporting will stop after the fifth report. One popular way to run the `vmstat` command is to leave out the count parameter; `vmstat` will then generate reports continuously until the command terminates.

Except for the avm and free columns, the first report contains per-second average statistics since system startup. Subsequent reports contain statistics collected during the interval since the previous report.

The following detailed description of the columns conveys useful information about CPU utilization:

kthr: Kernel thread state changes per second over the sampling interval.

- **r:** Number of kernel threads placed in run queue per second averaged over the sampling interval. This will be zero on an idle system. The higher the number, the busier the CPU. A number that is consistently higher than the number of processors in the system indicates a CPU bottleneck.
- **b:** Average number of kernel threads placed in the Virtual Memory Manager (VMM) wait queue; for example, awaiting for Journaled File System (JFS) I/O.

cpu: Breakdown of percentage usage of CPU time. For multiprocessor systems, CPU values are global averages among all processors. Also, the I/O wait state is defined system-wide and not per processor.

- **us:** Average percentage of CPU time executing in the user mode.
- **sy:** Average percentage of CPU time executing in the system mode.
- **id:** Average percentage of time that CPUs were idle and the system did not have an outstanding disk I/O request.
- **wa:** Average percentage of CPU time that the CPUs were idle during which the system had an outstanding disk I/O request. This value may be inflated if the actual number of I/O requesting threads is less than the number of idling processors.

The total of these four columns should be 100%, or very close. If the sum of user and system (us and sy) CPU-utilization percentages consistently approaches 100%, the system may be encountering a CPU bottleneck.

iostat

The `iostat` command is used primarily to monitor system input and output devices, but it can also provide CPU utilization data. The sample in Figure 6 is generated on an AIX 4.3.2 system by entering `iostat 5 3`.

```
kthr     memory             page                   faults           cpu
-------- ------------------ ---------------------- ---------------- -----------
r  b     avm   fre  re  pi  po  fr  sr  cy     in    sy   cs   us  sy  id  wa
0  0   27726  1677   0   0   0   0   0   0    188  1380  157   57  32   0  10
2  1   27726  1609   0   0   0   0   0   0    214  1476  186   48  37   0  16
2  1   27726  1582   0   0   0   0   0   0    248  1470  226   55  36   0   9
3  1   27726  1465   0   0   0   0   0   0    238   903  239   77  23   0   0
2  1   27726  1445   0   0   0   0   0   0    209  1142  205   72  28   0   0
```

Figure 5. vmstat report

```
tty:      tin      tout     cpu:   % user    % sys    % idle   %iowait
          0.0       4.3                0.2      0.6      98.8       0.4

Disks:    % tm_act     Kbps       tps    msps   Kb_read   Kb_wrtn
hdisk0         0.0      0.2       0.0              7993      4408
hdisk1         0.0      0.0       0.0              2179      1692
hdisk2         0.4      1.5       0.3             67548     59151
cd0            0.0      0.0       0.0                 0         0


tty:    tin      tout     cpu:   % user    % sys     % idle  %iowait
        0.0      30.3                8.8      7.2       83.9      0.2

Disks:   % tm_act     Kbps       tps    msps  Kb_read   Kb_wrtn
hdisk0         0.2      0.8       0.2                4         0
hdisk1         0.0      0.0       0.0                0         0
hdisk2         0.0      0.0       0.0                0         0
cd0            0.0      0.0       0.0                0         0

tty:   tin      tout      cpu:   % user    % sys     % idle  %iowait
       0.0       8.4                0.2      5.8        0.0     93.8

Disks:   % tm_act     Kbps       tps    msps  Kb_read   Kb_wrtn
hdisk0         0.0      0.0       0.0                0         0
hdisk1         0.0      0.0       0.0                0         0
hdisk2        98.4     75.6      61.9              396      2488
cd0            0.0      0.0       0.0                0         0
```

Figure 6. iostat report

Just like the report generated by the `vmstat` command, the first report contains statistics averaged since the system started up. Subsequent reports contain statistics collected during the interval since the previous report.

Four columns that show the breakdown of CPU usage time conveys the same information as the `vmstat` command, and should total approximately 100%. If the sum of user and system (us and sy) CPU-utilization percentages consistently approaches 100%, the system may be encountering a CPU bottleneck.

sar

The `sar` command has two forms: the first form will sample, display, and/or save system statistics; the second form processes and displays previously captured data.

The `sar` command can provide queue and processor statistics just like `vmstat` and `iostat` commands, however it has two additional features. First, each sample has a leading time stamp, so an overall average appears at the end of the samples. Second, the `-P` option can be used to generate per-processor statistics in addition to the global averages among all processors.

Figure 7 shows sample output from a four-way symmetric multiprocessor (SMP) system that resulted from entering two commands:

◆ `sar -o savefile 5 3 > /dev/null &`

Note: This command collects the data three times at five-second intervals, saves the collected data in `savefile`, and redirects the report to null so that no report is written to the terminal.

◆ `sar -P ALL -u -f savefile`

Note: The `-P ALL` is specified to get per-processor statistics for each individual processor, `-u` gets CPU usage data, and `-f savefile` tells `sar` to generate the report using the data saved in `savefile`.

Improving System Performance

For a CPU-bound system, you can improve the system performance by manipulating thread/process priorities of a specific process

or tuning the scheduler algorithm to set a different system-wide scheduling policy.

Changing User-Process Priority Using the nice Command

The standard `nice` value of a foreground process is 20; the standard `nice` value of a background process is 24. The system uses the `nice` value to calculate the priority of all threads associated with the process. Using the `nice` command, a user can specify an increment or decrement to the standard `nice` value so that a process can be started with a different priority. The thread priority is still non-fixed and gets different values based on the thread's CPU usage.

Any user can run a command at a lower-than-normal priority by using `nice`. Only root can use `nice` to run commands at higher-than-normal priority.

For example, the command `$ nice -5 iostat 10 3 >iostat.out` causes the `iostat` command to start with a `nice` value of 25 (instead of 20), resulting in a lower starting priority. The values of `nice` and priority can be viewed using the `ps` command with the `-l` flag.

Figure 8 shows a typical output using the `ps -l` command.

As root, you can run `iostat` at a higher priority with `# nice –5 vmstat 10 3 >io.out`. The `iostat` command can run with a `nice` value of 15, resulting in a higher starting priority.

The renice command. If a process is already running, you can use the `renice` command to alter the `nice` value, and thus the priority. The processes are identified either by process ID, process group ID, or the name of the user who owns the processes. `renice` cannot be used on fixed-priority processes.

```
AIX saturn2 3 4 005045484C00    03/31/99

16:02:54     cpu     %usr    %sys    %wio   %idle
16:02:59       0       45      45       5       5
               1       25      67       3       5
               2       59      36       1       4
               3       48      41       4       7
               -       44      48       3       5
16:03:04       0       43      43       9       5
               1       27      65       3       5
               2       51      40       5       4
               3       48      41       4       7
               -       42      48       5       5
16:03:09       0       35      47      10       8
               1       25      63       9       3
               2       55      44       1       0
               3       41      41      11       7
               -       39      49       8       4

Average        0       41      45       8       6
               1       26      65       5       4
               2       55      40       2       3
               3       46      41       6       7
               -       42      48       5       5
```

Figure 7. sar report

The nice() or setpriority() Subroutines

There are now two system calls which allow users to make individual processes or threads to be scheduled with fixed priority. The `setpri()` system call is process-oriented and `thread_setsched()` is thread-oriented. Use caution when calling these two subroutines, since improper use may cause the system to hang.

An application that runs under the root user ID can invoke the `setpri()` subroutine to set its own priority or the priority of another process. The target process will be scheduled using the `SCHED_RR` scheduling policy with a fixed priority. The change is applied to all the threads in the process.

```
retcode = setpri(0, 45);
```

gives the calling process a fixed priority of 45.

```
F       S   UID    PID   PPID  C  PRI  NI  ADDR   SZ     WCHAN    TTY   TIME     CMD
240001  A     0  23082  20266  0   60  20  7bfe  484           pts/2   0:00     ksh
8200011 A     0  23634  23082  0   70  25  35cc  112  50c86458 pts/2   0:00  iostat
```

Figure 8. Using ps -l to observe process priority

```
retcode = setpri(12345, 35)
```

gives the process with PID of 12345 a fixed priority of 35.

If the change is intended for a specific thread, `thread_setsched()` subroutine can be used.

```
retcode = thread_setsched(thread_id,
priority_value , scheduling_policy)
```

The parameter `scheduling_policy` can be one of the following: `SCHED_OTHER`, `SCHED_FIFO`, or `SCHED_RR`. When `SCHED_OTHER` is specified as the scheduling policy, the second parameter `priority_value` is ignored.

Change the Scheduling Algorithm Globally

AIX 4.0 has an enhancement that allows users to make changes to the priority calculation formula using the `schedtune` command. This enhancement is available in PTF U435266 for AIX 3.2.5 users.

Adjusting r and d. As mentioned earlier, the formula for calculating the priority value is as follows:

```
Priority = x_nice + (C * r/32 * x_nice_factor)
```

The recent CPU usage value is displayed as the "C" column in the `ps` command output. The maximum value of recent CPU usage is 120. Once every second, the CPU usage value for each thread is degraded using the following formula:

```
New Charge C = (Old Charge C) * d / 32
```

The default value of `r` is 16; therefore, the thread priority is penalized by `recent CPU usage * 0.5`). The `d` also has a default value of 16, which means the recent CPU usage value of every process is reduced to half of its original value once every second.

For some users, the default values of `r` and `d` do not allow enough distinction between foreground and background processes. These two values can be tuned using `-r` and `-d` options to the `schedtune` command. For example:

```
# schedtune -r 0
```

(`R`=0, `D`=.5) indicates that the CPU penalty was always 0. The priority value of the process would effectively be fixed, although it is not treated like an RR process.

```
# schedtune -d 32
```

(`R`=0.5, `D`=1) indicates that long-running processes would reach a C value of 120 and stay there. The recent CPU usage value does not get reduced once every second, and the priority of long-running processes would not fluctuate back to low numbers (higher importance) to compete with new processes.

Changing the time slice. Although the `schedtune` command can modify the length of the scheduler time slice, the time slice change only applies to RR threads. This does not affect threads running with other scheduling policies. The syntax for this command is:

```
schedtune -t n
```

where `n` is the number of 10ms clock ticks to be used as the time slice. `schedtune -t 2` would set the time slice length to 20ms.

You must log on as root to make changes using the `schedtune` command. The changes will not persist across system boots. Adding the command to the end of the `/etc/inittab` file could ensure that your changes stay effective after rebooting.

More Help

Other techniques that can help a CPU-bound system include the following:

Scheduling. Depending on the relative importance of applications, you could schedule less important ones for off-shift hours using `at`, `cron`, or `batch` commands.

mkpasswd. If your system has thousands of entries in the `/etc/passwd` file, you could use `mkpasswd` command to create a hashed or indexed version of the `/etc/passwd` file to save CPU time spent in looking up a user ID.

Tuning individual applications. The `ps` command or profiling can identify an application that is consuming large fractions of CPU time. This information can then be

used to narrow the search for a CPU bottleneck. Once the problem area is found, the application can be tuned up or improved. This may involve recompiling the application and/or changing the source code.

Upgrade. Upgrading the system to a faster CPU—or more CPUs—may be necessary if none of the tuning improves the performance.

Case Studies

The following real-world examples show how these theories and techniques were implemented.

Case 1

Symptoms: The user has a batch script that starts up 500 other batch scripts and each of these scripts queries and updates a database. All scripts also start as a client request from another machine. Each client request creates a database user thread on the database server machine. The response time began at less than 10 seconds for a period of time. Then the response time gradually became worse; at times it was more than a minute, sometimes two minutes.

Diagnosis: The run queue began growing until it reached into the hundreds. Another symptom included the CPU being 100% utilized (this was an eight-way SMP system), with 99% in user mode. By examining an AIX trace sample collected for a few seconds, a pattern emerged. While a thread was using the CPU, a network packet would arrive and cause a network adapter interrupt. This would take the currently running thread off its CPU so the interrupt could be serviced.

After servicing the interrupt, the scheduler checks to verify if any other threads are runnable and if they have a better priority than the currently running thread. Since the currently running thread had run for a few time slices already, its CPU priority had increased as it accumulated CPU ticks.

Each of the 500 scripts began with priority 60. If they were runnable, they would preempt any currently running thread if that thread's priority was higher than 60. The preempted thread would then be put at the end of the run queue and have to wait for the CPU until its priority rose again.

One effect of this preemption was that sometimes a thread would be preempted while holding a database lock. Since this type of lock is implemented at the application layer within the database software, the kernel does not know that the thread is holding a lock. If the lock was a kernel-level lock or a `pthread` library mutex lock, then the kernel could perform priority boosting and boost a thread's priority to the same level as that of a running thread that is requesting the lock. This way, the requesting thread will not have to wait long for the lock holder to get the CPU again and release the lock.

Since the lock in this scenario was a user lock, the database thread would spin on the lock until it exhausted its spin count (a tunable database parameter) and then go to sleep. So the 99% used CPU was mostly due to the threads spinning on database locks.

Prescription: After determining that priority preemption was having a negative effect, we tuned the scheduler formula, which calculates the thread priority. This particular formula was `pri = base_pri + NICE + (C * r/32)` where `pri` is the new priority, `base_pri` is 40, `NICE` is the `nice` value (20 in this case), C is the CPU usage in ticks, and `r` is 16.

As a thread accumulates CPU ticks, its priority value becomes larger, thereby making its priority lower.

The `schedtune` command provides a way to change the value of `r` by using the `-r` flag. Running `schedtune -r 0` causes `r` to be 0, which then makes the CPU penalty factor `(C * r/32)` 0. This prevents priorities from changing unless the `nice` value is changed. If the `nice` value was the same for all threads, then threads can complete their time slices without being preempted due to priority changes. This allows the thread that is currently running and holding the database lock to keep running, then release the lock.

Results: This had an instantaneous impact on the performance. The response time, which was over two minutes by this time, started getting better until all of the scripts were completing in just a few seconds.

The C value in the priority formula is recalculated once a second by a CPU usage decay factor (`C = C*d/32`). Setting the `d` value to 0 using the `schedtune` command would have accomplished the same result. In this case, if `d` =0, then `C*d/32 = 0`. Since the CPU penalty factor is `C*r/32`, this also becomes 0 so that the priority will be just 40 + `NICE`.

Case 2

Symptoms: An RS/6000 was used as both a database and an application server. Users input requests into a forms-based application, then submitted transactions. Users noticed that at certain times the forms would take longer to get updated on their screens, or their usual short-running queries would return in a longer time period.

Diagnosis: When this slowness was observed, there were also some long-running database batch jobs that were submitted to the system. Normally, such batch jobs would be run at night, but near the end of the month, additional batch jobs were run during the day as well while the users were on the system. The batch jobs were CPU-intensive and constantly on the run queue. Therefore, users' threads had to compete with the threads of the batch jobs for the CPU.

With priorities degrading as CPU usage increased, the batch jobs' priorities became worse and allowed the users' threads to run. However, once a second, the kernel decays the CPU usage value (C) by half. This allowed the priorities of the batch jobs to improve in a short time period. So the batch jobs would again compete for the CPU with the users' threads.

Prescription: By changing the decay factor (`d/32`) used to reduce CPU usage once a second, we improved performance for the users. We used the `schedtune` command to set the `d` value to 31. The higher the value of `d`, the higher the value of C (`C=C*d/32`). Since C is used to calculate priorities (`pri=40+NICE+C*r/32`), then as C became larger, the priority would get worse. By setting the `d` value to a higher number, the C value is reduced at a slower than usual rate.

Results: The users' threads get the CPU more often than the batch threads, and the users saw an immediate improvement in performance. Of course, the batch jobs would be slowed down somewhat, but these jobs would get the CPU whenever the users had any "think" time or had to wait on I/O. The impact was minimal on the batch jobs, but performance improvement for the users was dramatic.

Tips and Techniques

Sometimes a process requires only a few processors in an SMP system. An example is the need to know the scalability of an application running on an SMP ranging from one to the maximum number of processors. One way to do this is to use a system command such as `cpu_state` to disable one or more processors. Although this method sounds easy, it requires a reboot, and the `cpu_state` command applies only to Micro Channel® Architecture (MCA) machines.

A faster and more flexible alternative is to take advantage of the CPU scheduling policy to allow a dummy job to "occupy" the unwanted processors. Follow these steps.

1. Write a simple, infinite loop program.
2. Set it to a high priority using the `setpri()` API. It is changed to an RR policy job at a fixed, higher priority.
3. Start this job and use the `bind_processor` command to bind it to a particular processor.

Once the job is done, you now have one less processor for the application to use.

Be careful when you use this technique. Use a priority only high enough to block the application under test; do not go too high. We accidently occupied the last processor in an SMP box, which required the entire system to be rebooted.

Improve System Performance

A final tip describes some odd things that impact performance. During one of our benchmarks, we noticed that the CPU usage reached 100% with most of the time being charged to "system." At that time, the application performance degraded noticeably.

After we collected an AIX trace, we noticed a repeating pattern. One application process would encounter a page fault on an address. That page fault caused a protection exception in the VMM, which in turn caused the kernel to send this process a `SIGSEGV` (segmentation violation) signal.

When the process resumed, the page faulted on the same address again, which then caused yet another protection exception and another `SIGSEGV` signal to be sent to the process. The default signal disposition for the `SIGSEGV` signal is to kill the process and generate a core dump—but in this case, the application continued on and stayed in this loop. Most of the CPU time was spent in this loop.

After a massive investigation, we discovered the problem: A developer for another component had installed a signal handler to catch the `SIGSEGV` signal in the code during the test process. After the testing was completed, the developer had forgotten to remove the signal handler. That component then linked with the rest of the application and, during the benchmark, another unrelated component of the application caused a segmentation fault. This old signal handler caught the exception, ignored it, and caused the process to resume. The current instruction (the one which caused the exception) was then restarted, causing an infinite loop to occur.

***Lee Cheng**, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Ms. Cheng currently works as a senior consultant for RS/6000 and AIX software vendors. She provides support to them in the areas of application benchmarks, performance tuning, application porting, and internationalization. Before joining the RS/6000 ISV Technical Support group, she was a developer for compilers and the AIX system management component. She holds an MS in Computer Science from the University of Kentucky.*

***Wayne Huang**, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Huang is a senior consultant for RS/6000 and AIX systems with a focus on banking, finance, and securities industries. He provides AIX support to ISVs in the areas of application design, problem determination, system performance tuning, and application benchmarks. Mr. Huang holds a BS in Physics from National Taiwan University and an MS in Computer Science from the University of Texas at Austin.*

***Matt Accapadi**, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Accapadi is a software engineer responsible for performance of AIX and Oracle on AIX, and works with other vendors to improve their application performance on AIX. He is also responsible for solving performance issues on customer systems, tuning benchmarks for optimal performance, and teaching courses on AIX performance tuning. Mr. Accapadi has a BS in Computer Science from Texas A&M University.*

GlobalHost for AIX

By George Kraft IV

GraphOn's GlobalHost for AIX provides a super lightweight Java applet and network protocol for using the X Window System with thin clients.

With the advent of thin-client computing comes the need to support existing user interfaces from IBM server platforms. UNIX®, OS/400®, and OS/390® can use Telnet, 5250, and 3270 character-based terminal emulators that are implemented in Java™. In contrast, the Microsoft® Windows®, the X Window System®, and the Macintosh® graphical user interfaces are entire windowing environments not natively supported by some thin clients.

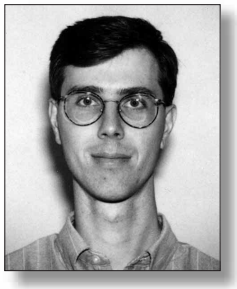
The X Window System, also known as X or X11, was widely released to the public in 1985, and since that time it has become the *de facto* standard graphical user interface (GUI) infrastructure for open-system UNIX platforms. The Massachusetts Institute of Technology standardized the application programming interface (API) for X11 software in 1987 with the release of X Version 11, Release 3 (X11R3). More recently, the X.Org Consortium announced the availability of the X11R6.4 release.

Based on a network protocol rather than procedural calls, X11 enables distributive work to be performed in a heterogeneous network environment. This can occur because graphical X11 client applications running anywhere on the network can access the X Server's display device. A GUI such as Motif® is built on X11 to ensure stylistic consistency for windows, menus, and other graphical components.

The client X11 libraries and the X Server are key components of the X Window System. Graphical X11 client applications are programs built upon the X11 and Motif libraries that perform tasks for graphical presentation, such as terminal emulation, spreadsheet, word processing, and window management. Graphical X11 client applications typically run on multitasking host computers that send requests to an X Server—that is, to the same machine or to a remote location on a network machine—to be viewed by the end user.

The X-Server software includes device drivers for the video card, keyboard, and mouse. The X Server distributes user input to, and accepts output requests from, various X-Client application programs through the X11 protocol, as shown in Figure 1. X11 client applications interact with the X Server by requesting manipulation of resources: windows, cursors, fonts, pixmaps, and graphic contexts. Because X11 is more than just the X protocol, thin clients must support the X Server's full implementation. Therefore, an X Server must be part of the thin-client architecture to support the X Window System, which includes the following:

- X Resource Manager
- ICCCM
- X Properties/Atoms
- X-Tree List
- Window Manage Protocol



George Kraft IV

The sample X Server was initially written in C and has matured over the past decade. It could be rewritten in Java or embedded in the thin client's operating system; however, doing either would burden the system's precious memory resources. Therefore, most X11 solutions developed for thin clients retain the X Server in its native environment on a UNIX host and render it to a Java applet viewer on a thin client.

GlobalHost

GraphOn™ Corporation has used the X Server's modular layering by replacing the device-driver layer (ddx) that normally speaks to the hardware with a proxy that communicates with a thin-client applet viewer. Its GlobalHost™ X Server is essentially the same server, but it contains a virtual graphics card, or canvas that is remotely located on the thin client.

GraphOn's GlobalHost behaves like a standard 8-bit pseudo-color X Server. Instead of driving a graphics card, it proxies the graphical requests remotely via the RapidX™ protocol to a thin-client applet viewer, shown in Figure 2. All of the basic X11 enablement remains intact on the server. The AIX® libraries, fonts, clients, configuration files, and system management are the same as those used for a local X Server on the system's bitmap display or for a remote X Terminal.

RapidX

The GlobalHost X Server and the thin-client Java applet viewer communicate using the RapidX protocol. RapidX reduces network traffic by approximately a factor of seven, as `netstat` indicates, by using three layers of compression.

In the first layer, the GlobalHost X Server sends less data over the network by translating X11 primitive to RapidX primitive, for example, by drawing a polyline. The second layer uses two-dimensional data compression that typically reduces larger bitmaps and fonts up to 60%.

Finally, general compression, similar to the Limpel-Ziv algorithm commonly used on UNIX systems, is applied to the data flow. This data-flow compression results in improved efficiency on low-bandwidth wide area network (WAN) and dial-up connections where network throughput is critical.

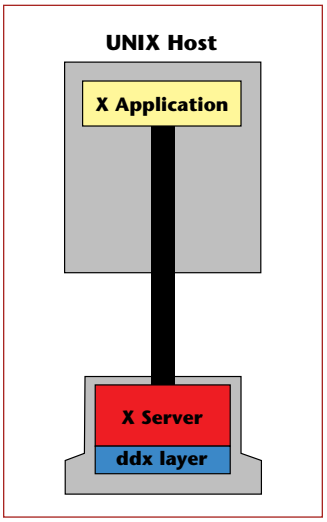


Figure 1. X Server and networked machine

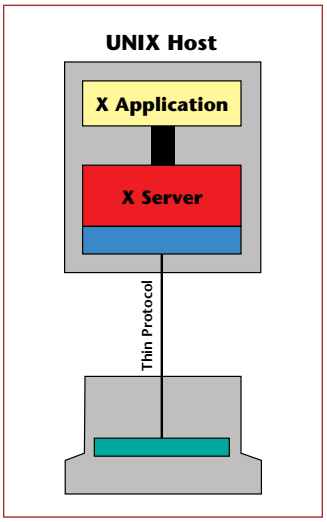


Figure 2. GlobalHost X-Server

GO-Joe

At the other end of the RapidX socket is the GO-Joe™ Java applet viewer for thin clients. The thin-client GO-Joe applet viewer handles keyboard and mouse input, and displays the X Window System environment in a browser window or in the applet viewer.

To start the GlobalHost X Server and the GO-Joe Java applet, the thin client is either preconfigured or the end user surfs to a GO-Joe applet HyperText Meta-Language (HTML) page. The HTML page instructs the Java Virtual Machine (JVM) on the thin client to download the GO-Joe applet and to authenticate the user. Once the user is authenticated to the host AIX server, GlobalHost is started together with the users normal X11 environment. The end user does not need to learn anything new.

GO-Joe's approximate 300 KB size represents a substantial memory-resource savings over a full X Server in a Java-only thin-client or in a fat-client PC environment. In a PC client X-Server setup, tens of megabytes worth of libraries, bitmaps, and fonts would be installed on each and every system. In the GlobalHost and GO-Joe environment, everything is centrally managed on the server.

Overall, the impact of GlobalHost and GO-Joe on the host server should be similar to that of an X Terminal. The X11 applications run on the host server and are displayed to the client device. The GlobalHost X Server would use runtime resources of the host system like any other server application.

Interoperability

The IBM RS/6000® system AIX 4.3 Bonus Pack will deliver GlobalHost Version 1.0.8.0 and GO-Joe Version 2.2.0.0 to provide X-Client application enablement to Java-enabled devices for which the X Window System is not native. GraphOn has also licensed its GO-Joe technology to Sun Microsystems®. These offerings allow AIX and Solaris® to provide interoperability between the UNIX server platforms and various Java-enabled clients. In addition, the GlobalHost also can serve the native Win32 GO-Global™ application running on Windows 95.

For more information, visit *http://www.graphon.com*

***George Kraft IV**, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Kraft is an advisory software engineer for IBM's Network Computer Division. He recently moved from IBM's RS/6000 Division where he worked on the AIX integration of the IBM Network Station™. He has a BS in Computer Science and Mathematics from Purdue University.*

Linking and Loading in AIX 4.3

By Gary R. Hook, Vandana Kumar, and Steve Peckham

This article provides an overview of various aspects of the AIX linker and system loader important in developing new applications on AIX and in porting existing UNIX applications from other platforms to AIX. It shows several examples, including source code, makefiles, suggested command-line options, and other details.

In the past, program compilation was always limited to static linking. This technique combined together multiple libraries and multiple object files along with their code and data, and created an executable file. This single and often very large file contained all the symbol definitions within it. Although this made the linking and execution process fast and simple, every running process had to load into memory its own private copy of every library function it used.

The need to share libraries that were already loaded in memory by other programs led to the concept of *shared libraries*. When a program is linked with a shared library, the code within the shared library is not included in the generated executable. Instead, enough information is saved in the program to find and load the library when needed.

Shared library code is loaded into global system memory by the first program, then it is shared by all subsequent programs. This technique reduces memory usage and loading time. Additionally, enhancements to the shared libraries can be added without relinking the executable.

This approach works well if the location of symbol definitions is known when the program is initially linked. When the location of some symbol definitions is not known at link time or if alternate definitions are to be provided for some functions, a more flexible linker is needed. AIX® 4.3 provides this flexibility via its *runtime linker*, which allows programs to dynamically load shared objects into the running process via the `libdl.a` routines.

This article describes the new linking and loading features available in AIX 4.3, including code examples.

Symbol Resolution in AIX 4.3

Without the runtime linker in AIX, all symbols must be accounted for when the module is linked. The undefined symbols must be listed in the module's import list or be deferred. Symbols are deferred if they are listed as being defined by `#!` in the import list, which provides the name of the defining module for each listed symbol. These symbols are resolved during load-time.

Deferred Symbols

The linker and loader effectively ignore deferred symbols because they are resolved later. This is usually done by dynamic

Gary R. Hook

Vandana Kumar

Steve Peckham

loading of the defining module into the running process by the `load()` or `dlopen()` calls. All modules that are opened in the address space of the executing process cause the outstanding deferred imports to be resolved. This happens according to the scope of the flags specified in the `dlopen()` and `dlsym()` function calls. Note that the defining module has to export the referenced symbols for the referencing module to find them.

When using the runtime linker, however, symbols can be left undefined at link-time. These symbols are marked as being imported from ". ." in the module's symbol table, which can be listed by using the `dump -HTv <module name>` command. The runtime linker resolves these symbols later. Import lists are not needed in this case.

The functioning of the deferred symbols does not change under runtime linking. All deferred symbols must be listed as defined by `#!` in the import file. The runtime linker ignores these symbols when they are designated as deferred. If no deferred symbols are identified, the linker treats all undefined symbols as ". ." imports, and the runtime linker searches and resolves these symbols at load-time. If the runtime linker is unsuccessful in locating the definition of any of the referenced symbols, the module fails to load and the program exits.

Deferred imports (also called anonymous imports) are listed as being imported from `[noIMPid]` in the symbol table section of the `dump -HTv <module >` output.

Third-Party Plug-in Software

Programs often provide stubs for third-party plug-in software so that some function symbols referenced by the program are executed only if the plug-in software is available. If this software is unavailable, the symbols are referenced but never get defined or used.

The runtime linker cannot handle these symbols because the load-time search among all exported symbol definitions yields nothing. Therefore, they should be marked as deferred to get past the loading phase. Once the module is loaded, the deferred symbols must be defined before the functions in question are ever called, otherwise they can be left undefined. If a call is ever made to an unresolved deferred import, the application will terminate with a segmentation fault in the glink code (glue code for branching between modules).

Dealing with undefined symbols that do not have a definition at module load-time requires some extra work. However, it ensures symbol availability in advance for better execution performance. Note that for the best performance on any platform, the symbols under runtime linker control should be kept to a minimum.

In AIX, libraries are often archives of multiple object files, which could reference one another or have dependencies of their own.

Shared Modules

Generally in AIX, a shared module used as input is listed only if a symbol in the shared module is actually referenced. If so, it would be listed in the loader section of the symbol table of the output file. When the runtime linker is used, however, it might be necessary to have the shared modules listed in the loader section, even if there are no symbols referenced.

By using runtime linking (via the `-brtl` option), all shared modules (those that are not archive members) on the command line are listed as dependents (preserving the command-line order). The system loader loads all such shared modules when the program runs, and the symbols exported by these shared modules are used by the runtime linker. Shared modules that are members of archives are listed as dependents only if they are referenced. The linker does not assume that archive members will be needed. Some archive members may have requirements that cannot be satisfied at program load-time, thereby causing loading failure.

In AIX, system libraries are often archives of multiple object files, which could reference one another or have dependencies of their own (which may or

may not be available at runtime). Also, all archive members of system libraries are not necessarily shared objects. For this reason, it is not enough to simply add a reference to them in the main executable. All required system libraries must be linked in to build each shared module. By linking with system libraries, all local definitions of symbols get resolved within the shared module.

To force a module in an archive to be loaded, an import must be used with the *autoload* keyword and the member name. For example, to enable autoloading for an archive member `foo.so`, create a file with the following lines, and add the file as a member to the archive:

```
# autoload
#! (foo.so)
```

For the module load to be successful, all its dependent modules must also load successfully. Therefore, if a shared module references symbols defined by other shared modules, those modules also must be loaded to resolve symbols referenced by the first module. This continues until no more modules need to be loaded. Only deferred imports do not cause a search and load of the defining module at load-time.

The next section explores the runtime linker in more detail, including code examples.

Runtime Linking

Runtime linking is the ability to resolve symbols in shared modules after the program has begun to execute. It is a mechanism that provides runtime definitions and symbol rebinding capabilities.

Note that the main application must be built to enable runtime linking. Simply linking a module with the runtime linking capability is not sufficient. This structure allows a module to be built to support runtime linking (such as third-party modules), yet continues to function in an application that has not been so enabled. See Figure 1.

The option `-G`, which is used to enable runtime linking in shared modules, is a combination of the options found in Figure 2.

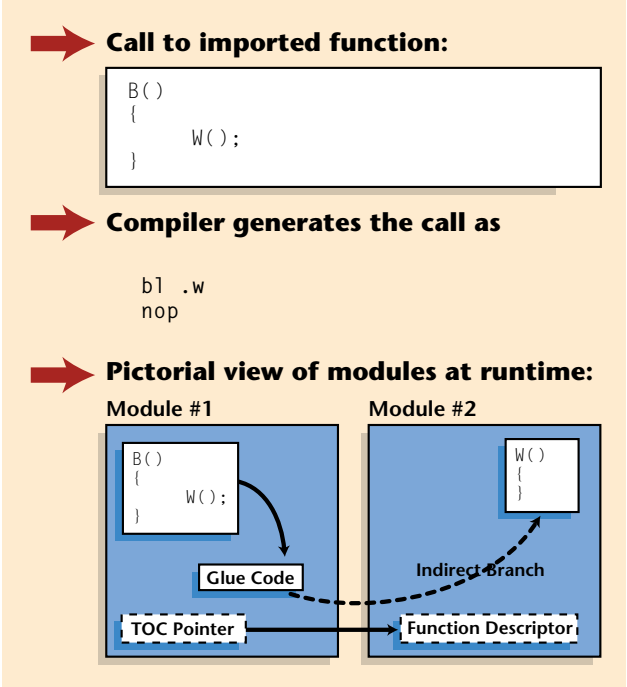


Figure 1. Generated code

`-berok`	Enables creation of the object file even if there are unresolved references.
`-brtl`	Enables runtime linking. All shared objects listed on the command line (those that are not part of an archive member) are listed in the output file. The system loader loads all such shared modules when the program runs, and the symbols exported by these shared objects may be used by the runtime linker.
`-bnosymbolic`	Assigns this attribute to most symbols exported without an explicit attribute.
`-bnortllib`	Removes a reference to the runtime linker libraries. This means that the module built with `-G` option (which contains the `-bnortllib` option) will be enabled for runtime linking, but the reference to the runtime linker libraries will be removed. Note that the runtime linker libraries should be referenced to link the main executable only.
`-bnoautoexp`	Prevents automatic exportation of any symbol.
`-bM:SRE`	Build this module to be shared and reusable.

Figure 2. The -G option

The option `-brtl` alone should be used to enable runtime linking in main executables.

For a simple introduction to runtime linking, see Example 1 in this article. In this example, the runtime linker searches and locates the undefined symbols during program execution. This eliminates the need for programmers to track which module calls and defines which symbols, or to keep the export/import lists updated.

In general, to take advantage of the AIX architecture, the shared modules should be built to be as self-contained as possible. Note that AIX performance and efficiency is best exploited by a well-organized application structure with a well-defined interface between modules.

Rebinding Symbols

It is often necessary to rebind function symbols so that a new definition for them may be used instead of the definition from which the module was originally built. In this case, the main program must be linked with the runtime linker, and the module providing the alternate definition must export the symbol and must be listed before the original module on the command line.

Based on how the module was linked, some references to exported symbols are bound at link-time and cannot be rebound. For example, if a symbol is referenced in the same file that also defines it, the symbol will be bound at link-time and cannot be rebound.

When runtime linking is used, a reference to a symbol in the same module can only be rebound if the symbol is exported with the proper attributes. References to symbols with the "symbolic" attribute cannot be rebound. References to symbols with the "nosymbolic" attribute can be rebound if the symbols are variables. For function symbols, calls using a function pointer can be rebound, but direct function calls cannot. The "nosymbolic-" attribute, which is the default, is provided for compatibility with previous versions of AIX.

When a symbol is rebound, a dependency is added from the module using the symbol to the module defining the symbol. This dependency prevents modules from being removed from the address space prematurely. This is important when a module loaded by the `dlopen()` subroutine defines a symbol that is still being used when an attempt is made to unload the module with the `dlclose()` call.

It is also possible to rebind the definitions of system library symbols. For example, an application can define its own `malloc()` to override or extend the functionality of the system-defined `malloc()` in `libc.a`. For this, the defining shared module (in this case `shr.o` in `libc.a`) has to be rebuilt using `rtl_enable` so that the references internal to the module become hookable. Once this is done, the runtime linker can override all references to `malloc()`. Without rebuilding `libc.a via rtl_enable`, the internal workings of `libc.a` remain "tight" and cannot be rebound.

New Features

AIX 4.3 provides several new linker/loader options. These options, which give additional programmatic control over the behavior of the modules, are listed and described briefly in this section.

> ***When runtime linking is used, a reference to a symbol in the same module can only be rebound if the symbol is exported with the proper attributes.***

libdl.a routines: Provides the standard routines `dlopen()`, `dlclose()`, `dlsym()`, and `dlerror()` in `libdl.a`. These routines can be used with or without runtime linking. A new flag, `RTLD_MEMBER` can be passed to `dlopen()`, which allows an archive member to be loaded.

init/fini routines: Provides module initialization and termination routines via the `-binitfini` option.

Loading of archive members: Allows loading of archive members by passing the flag `L_LOADMEMBER` to `load()` or `loadAndInit()`. It may also be passed to `dlopen()`; the name of the flag is `RTLD_MEMBER` in the `dlopen()` case. This

function loads member `shr.o` from archive `lib1.a`. For example:

```
dlopen("lib1.a(shr.o)", RTLD_NOW|RTLD_MEMBER);
```

-bexpall option: Exports most symbols. Exports all global symbols (except imported symbols, unreferenced symbols defined in archive members, and symbols beginning with an underscore). Additional symbols may be exported by listing them in an export list. Any symbol with a leading underscore will not be exported by this option. These symbols must be listed in an exports list to be exported. Linking the module with the `makeC++SharedLib` script will also export these symbols. Note that the `makeC++SharedLib` script generates a complete export list for C++ modules by default, without specifying the `-bexpall` option in its command line.

Visibility attributes: Allows visibility specification of all symbols via the `-bsymbolic/-bnosymbolic/-bnosymbolic-` flags. This feature can be used to control which references can be rebound by the runtime linker. Visibility keywords also can be used in an import file to control the visibility of individual symbols.

-bdynamic/-bstatic: Controls whether shared objects used as input files should be treated as regular files. These options are toggles and can be used repeatedly in the same link line. When `-bdynamic` is in effect, shared objects are used in the usual way. When `-bstatic` is in effect, shared objects are treated as regular files. Additionally, when `-brtl` is specified and `-bdynamic` is in effect, the `-l` flag will search for files ending in `.so` as well as those ending in `.a`. Figure 3 shows an example.

-bautoexp option: Exports a symbol automatically if any command-line shared module imports the symbol from the special dot (`.`) file, or the module being linked contains a local definition of the symbol. The default is `-bautoexp`.

-bipath/-bnoipath options: Saves/discards the full path name of the shared module on the command line. The full path is saved in the loader section of the module. The `-bipath` option is the default.

```
cc -o main main.o -bstatic -lx -Lnewpath -bdynamic
-> libx.a treated as a regular object file
-> libc.a processed as a shared library

cc -o main main.o -brtl -lx -Lpath1 -Lpath2
-> Search path1/libx.so, then path1/libx.a
-> Search path2/libx.so, then path2/libx.a
```

Figure 3. -bdynamic/-bstatic

If `-bnoipath` is used, then only the base names will be saved in the output files loader section. For example:

```
cc -o main main.o dir/mylib.so /usr/lib/otherlib.a
```

This will cause the full specified path to be saved for `mylib.so` and `otherlib.a`. At load-time, the loader will always use these paths to find the shared objects. If the `-bnoipath` option was specified, only the base names would have been saved in the loader section.

-blazy option: Lazy loading is a mechanism for deferring the loading of modules until one of its functions is required to be executed. By default, the system loader automatically loads all of the module's dependents at the same time. By linking a module with the `-blazy` option, the module is loaded only when a function within it is called for the first time. Note that lazy-loading works only if runtime linking is not enabled. Also, only the modules referenced for their function definitions can be lazy-loaded.

Runtime linking: Consists of the `-G` and the `-brtl` options.

(`-G` option) Enables runtime linking. It is equivalent to the following: `-berok -brtl -bnortllib -bnosymbolic -bnoautoexp -bM:SRE`. These options can be overridden by subsequent options. The `-G` option is used to build runtime-enabled shared libraries, without actually linking in the runtime linker libraries.

(`-brtl` option) Enables runtime linking. It is equivalent to `-brtllib -bsymbolic`. The `-brtl` option is used to build runtime-enabled main executable and includes a reference to the runtime linker libraries in the link line.

Under the influence of runtime linking, all shared objects listed on the command line (those that are not archive members) are listed as dependent modules, which preserves the command-line order.

This option is also useful since it allows the `.so` files to be found with the `-l` flag. If the `.so` files are to be found but runtime linking is not needed, use `-brtl` and `-bnortllib` together.

"." imports: Marks the undefined imported symbols as ones that will be resolved by the runtime linker. If a `dump -HTv <shared module>` output contains "`. .`" as the IMPid for any symbol in the symbol table, the runtime linker will search for and resolve that particular symbol.

"." imports: Marks the undefined symbol as being defined in the main executable. For example:

```
#! .
foo
```

The runtime linker is not needed since the system loader resolves these "`.`" symbols. However, the main program must still export these symbols.

rtl_enable command: Relinks modules to enable runtime linking. The shared libraries shipped with AIX are not enabled for runtime linking, but they can be enabled by using this command. For example, if a program uses its own version of `malloc()`, a new instance of `libc.a` must first be created as follows:

```
rtl_enable -o /usr/local/lib/libc.a
/lib/libc.a
```

Then the program must be relinked:

```
cc ... mymalloc.o -L /usr/local/lib
-brtl -bE:myexports
```

In this example, `mymalloc.o` defines `malloc()` and `myexports` causes `malloc()` to be exported from the main program. Calls to `malloc()` from within `libc.a` will now go to `malloc()` defined in `mymalloc.o`.

-brtllib/-bnortllib: Enables or disables runtime linking. Default is `-bnortllib`. In general, it is recommended that runtime linking be enabled by linking in the `-brtl` option and not by the `-brtllib` option directly.

Tips and Techniques

The following tips and tricks can be used to help linking and loading on AIX 4.3.

- Shared modules that use any C++ code must always be linked using the `makeC++SharedLib` script. This script is a wrapper around the `ld` command and does some C++ related processing before linking the module. Check the compiler documentation for details on its use.

- The `dump -HTv <module name>` command provides information about the shared module's symbol table.

- The `genkld` command provides a listing of the currently loaded shared libraries. This command is part of the `perfagent.tools` fileset.

- The shared modules built in AIX 4.1.5 can be referenced by an AIX 4.3 binary if the symbols are not under the runtime linker control. The AIX 4.1.5 modules should be manually loaded into the executable's address space using `dlopen()` calls.

- Diagnose problems in case of a linking failure by using the binder options as follows:

  ```
  -bloadmap:<file>
  -bmap:<file>
  ```

- In AIX, libraries can be listed multiple times and in any order (except when symbols are rebound). The first occurrence of a symbol wins, although warnings appear on additional occurrences.

- At link-time for every shared module, all needed system libraries must be linked in, including (but not limited to)

  ```
  -lc -lC -lm -lX11
  ```

- If the `makeC++SharedLib` script is used to build shared libraries and export symbols, make sure that the system libraries are always specified with an `-l` option (for example, `-lc`) and never by name on the `script_s` command line. This allows the system libraries to be simply referenced

and not go through special C++ related processing.

- If large parts of the shared libraries are paged in all at once because of C++ calls or many references between libraries, it may be faster to read the library rather than demand-page it into memory. Remove read-other permission from all shared libraries and see if the loading performance improves. If it does, then reset the original permissions and set the following environment variable:

```
LDR_CNTRL=PREREAD_SHLIB
```

  By using this environment variable, the libraries are read very fast into the shared memory segment.

- If a loader error occurs at program execution time in which a certain library cannot be found (even though the `LIBPATH` clearly points to it), check the following:

  - The library should have read permission.
  - The loader section of the calling module's `dump` output should point to the correct directory.
  - The `-bnoipath` option should not be used with the library to link the module; all dependent modules of the library should be available for loading.
  - All dependent modules of the loading module have to load successfully, so follow the module tree to see which leaf did not load (possibly due to wrong `LIBPATH` or disabled read permissions).

- If linking or loading difficulties arise, it is best to simplify the environment (including disabling the version control system) and debug the problem by taking one step at a time.

Examples

This section begins with a few basic C files (shown in Figure 4), which will be used to illustrate several examples later in this section.

a.c

```
#include <stdlib.h>
#include <stdio.h>

extern int b();

int a()
{
    printf( "Now in function a()\n" );
    b();
    return( 0 );
}
```

b.c

```
#include <stdlib.h>
#include <stdio.h>

extern void c1();

int b()
{
    printf( "Now in function b()\n" );
    c1();
    return( 0 );
}
```

c1.c

```
#include <stdlib.h>
#include <stdio.h>

int c1()
{
    printf( "Now in function c1()\n" );
    return( 0 );
}
```

hello.c

```
#include <stdlib.h>
#include <stdio.h>

extern  int     a();

int main()
{
    printf( "\nHello World\n" );
    a();
    return(0);
}
```

Figure 4. Basic C files

Example 1

The example in Figure 5 shows building a shared library in AIX 4.3 using the runtime linker.

Figure 6 shows symbol table information for `liba.so` obtained by the command `dump -HTv liba.so`.

The loader symbol table output in Figure 6 means that the symbol `printf` comes from the shared module `shr.o`, an archive member of the system library `libc.a`. The runtime library will search for the symbol `b` when `liba.so` is loaded. This is indicated by the " . . " in the `IMPid` column for symbol `b` in Figure 6 (recall that runtime symbol lookup is only performed if the main application is built with runtime linking enabled).

Figure 7 shows how to build the final executable.

```
cc -c a.c -o a.o
cc -c b.c -o b.o
cc -c c1.c -o c1.o
cc -c hello.c -o hello.o

ld -o liba.so a.o -bnoentry -G -bexpall -lc
```

Figure 5. Build shared library with runtime linker

Loader Section

```
Loader Header Information
   VERSION#      #SYMtableENT      #RELOCent        LENidSTR
  0x00000001      0x00000003      0x00000006      0x00000023

  #IMPfilID       OFFidSTR        LENstrTBL       OFFstrTBL
  0x00000003      0x000000b0      0x00000000      0x00000000
```

Import File Strings

```
    INDEX           PATH            BASE           MEMBER
      0         /usr/lib:/lib
      1                           libc.a           shr.o
      2                             ..
```

Loader Symbol Table Information

```
[Index]     Value       Scn    IMEX Sclass     Type        IMPid             Name

  [0]     0x00000000    undef      IMP      DS EXTref     libc.a(shr.o)     printf
  [1]     0x00000018    .data      EXP      DS SECdef     [noIMid]          a
  [2]     0x00000000    undef      IMP      DS EXTref     ..                b
```

Figure 6. Symbol table information for liba.so

```
ld -o libb.so b.o -bnoentry -G -bexpall -lc
ld -o libc1.so c1.o -bnoentry -G -bexpall -lc
xlc -o hello liba.so libb.so libc1.so hello.o -L. -brtl

On executing "hello":

>hello

Hello World
Now in function a()
Now in function b()
Now in function c1()
```

Figure 7. Building the final executable

Example 2

Figure 8 shows a rebinding example.

f1.c

```
#include <stdio.h>
int func1()
{
    printf( "\tinside of func1()/%s...\n", __FILE__ );
    printf( "Calling func2()...\n" );
    func2();
}
```

f2.c

```
#include <stdio.h>

int func2()
{
    printf( "\tinside of func2()/%s...\n", __FILE__ );
    printf( "Calling func3()...\n" );
    func3();
}
```

f3.c

```
#include <stdio.h>

int func3()
{
    printf( "\tinside of func3()/%s...\n", __FILE__ );
    printf( "Calling func4()...\n" );
    func4();
}
```

f4.c

```
#include <stdio.h>

int func4()
{
    printf( "\tinside of func4()/%s...\n", __FILE__ );
}
```

main.c:

```
#include <stdio.h>

int func4() /* func4 is being redefined here in main.c */
{
    printf( “\tinside of func4()/%s...\n”, __FILE__ );
}

extern int func1();

main()
{
    printf( “Calling func1()...\n” );
    func1();
}
```

(continued on following page)

Figure 8. Rebinding example

(continued from previous page)
Compiling and linking:

```
xlc -D_POSIX_SOURCE -D_ALL_SOURCE -g -qextchk -c main.c
xlc -D_POSIX_SOURCE -D_ALL_SOURCE -g -qextchk -c f1.c
xlc -D_POSIX_SOURCE -D_ALL_SOURCE -g -qextchk -c f2.c
xlc -D_POSIX_SOURCE -D_ALL_SOURCE -g -qextchk -c f3.c
xlc -D_POSIX_SOURCE -D_ALL_SOURCE -g -qextchk -c f4.c
ld -o libshr2.so f3.o f4.o -G -bnoentry -bexpall
ld -o libshr1.so f1.o f2.o -lshr2 -G -bnoentry -L. -bexpall
xlc -o main main.o -lshr1 -lshr2 -L. -brtl -bE:main.exp
```

The following WARNING message will be emitted by the binder:

```
ld: 0711-224 WARNING: Duplicate symbol: .func4
ld: 0711-224 WARNING: Duplicate symbol: func4
ld: 0711-345 Use the -bloadmap or -bnoquiet option to obtain more information.
```

Since `main.o` appears before `libshr1.so` and `libsh2.so`, the `func4()` that is defined by `main` is used. The results of running `main` are as follows:

```
Calling func1()...
        inside of func1()/f1.c...
Calling func2()...
        inside of func2()/f2.c...
Calling func3()...
        inside of func3()/f3.c...
Calling func4()...
        inside of func4()/main.c...
```

Note that symbol `func4` has been rebound to the one defined in `main.c`.

Figure 8. Rebinding example

Example 3

Figure 9 shows an example of runtime linking and dynamic loading of non-linked objects.

main.c

```
#include <stdio.h>

int main()
{
    vmap_routine();
    usr_preempt();
}

void main_routine()
{
    printf( "in main_routine in main.c\n" );
}
```

(continued on following page)

Figure 9. Runtime linking and dynamic loading of non-linked objects

(continued from the previous page)

vmap.c

```
#include <stdio.h>

void vmap_routine()
{
    printf("in vmap_routine in vmap.c\n");
    main_routine();
    usr_routine();
}

void vmap_axs_routine()
{
    printf( "in vmap_axs_routine in vmap.c\n" );
}

void usr_preempt()
{
    printf( "in standard usr_preempt routine in vmap.c\n" );
}
```

usr.c

```
#include <stdio.h>
#include <dlfcn.h>

typedef void (*void_fn)();

void usr_routine()
{
    printf( “in usr_routine usr.c\n” );
    axs_routine();
}

void usr_preempt()
{
    void_fn routine;

    printf( “in correct usr_preempt routine in usr.c\n” );
    routine = (void_fn) dlsym( dlopen( “./libvmap.so”, RTLD_NOW ), “usr_preempt”);
    routine();
    routine = (void_fn) dlsym( dlopen( “./libext.so”, RTLD_NOW ), “ext_routine” );
    routine();
}
```

axs.c

```
#include <stdio.h>

void axs_routine()
{
    printf( "in axs_routine in axs.c\n" );
    vmap_axs_routine();
}
```

(continued on following page)

Figure 9. Runtime linking and dynamic loading of non-linked objects

(continued from the previous page)

ext.c

```
#include <stdio.h>

void ext_routine()
{
    printf( "in ext_routine in ext.c\n" );
}
```

Compile and link:

```
xlc -c -g usr.c axs.c ext.c main.c vmap.c -D_ALL_SOURCE
ld -o libusr.so usr.o -bnoentry -G -bexpall -bM:SRE -lc -ldl
ld -o libvmap.so vmap.o -bnoentry -G -bexpall -bM:SRE -lc
ld -o libaxs.so axs.o -bnoentry -G -bexpall -bM:SRE -lc
ld -o libext.so ext.o -bnoentry -G -bexpall -bM:SRE -lc
cc -o main main.o libusr.so libvmap.so libaxs.so -brtl -L.
```

Ignore the `ld` warning regarding duplicate symbols in this instance; there *are* two instances of `usr_preempt`. The output produced by this program is, as expected:

```
> main
in vmap_routine in vmap.c
in main_routine in main.c
in usr_routine usr.c
in axs_routine in axs.c
in vmap_axs_routine in vmap.c
in correct usr_preempt routine in usr.c
in standard usr_preempt routine in vmap.c
in ext_routine in ext.c
```

Figure 9. Runtime linking and dynamic loading of non-linked objects

This example is particularly interesting. First, symbol `usr_preempt()` is defined in two modules: `libvmap.so` and `libusr.so`. Second, symbol `usr_preempt()` in `usr.c` (`libusr.so`) uses the same symbol as defined in `vmap.c` (`inlibvmap.so`) via an explicit `dlopen()` of `libvmap.so`. Third, an explicit `dlopen()` of module `libext.so` is not linked to the main module.

Example 4

Figure 10 shows an example of calling an AIX 4.1.5 shared library from AIX 4.3 binary.

main.c

```
#include <stdio.h>
#include <dlfcn.h>

main()
{
    void *handle;
    void (*fct)();

    printf("hello from main\n");
```

(continued on following page)

Figure 10. Calling AIX 4.1.5 shared library from AIX 4.3 binary

(continued from the previous page)

```
    /* main() needs to run foo_() which is defined by libfoo.so.
     * libfoo.so has been previously built on AIX 4.1.5.
     */
    handle=dlopen("libfoo.so",RTLD_NOW);
    if (handle==NULL)
    {
        perror("dlopen");
        exit(1);
    }

    fct=(void (*)())dlsym(handle,"foo_");

    if (fct==NULL)
    {
        perror("dlsym");
        exit(1);
    }

    (*fct)();
    printf( "exit from main\n" );
}
```

foo.c

```
#include <stdio.h>
void foo_()
{
    printf( "hello from foo()\n" );
}
```

To build `main` on AIX 4.3:

```
xlc -brtl -g -o main main.o -L. -ldl
```

To build `libfoo.so` in AIX 4.1.5:

```
ld -o libfoo.so foo.o -bM:SRE -bE:exp.lst -bnoentry -lc
```

Where `exp.lst` contains the symbol `foo_`.

Running `main` yields the following output:

```
> main
hello from main
hello from foo()
exit from main
```

Figure 10. Calling AIX 4.1.5 shared library from AIX 4.3 binary

Example 5

Figure 11 shows autoloading of archive members.

main.c

```
main()
{
    bar();
}
```

bar.c

```
int bar()
{
    printf( "in bar()\n" );
    foo();
}
```

foo.c

```
#include <stdio.h>

int foo()
{
    printf( "in foo() which is correct...\n" );
}
```

barfoo.c

```
#include <stdio.h>

int foo()
{
    printf( "in barfoo() which is wrong...\n" );
}
```

foo.auto

```
#!
# autoload
#! (shr.o)
```

Makefile

```
all:        libbar.so libfoo.a main

main:       main.o
            cc -o main main.o -L. -lfoo -lbar -brtl

libbar.so:  bar.o barfoo.o
            ld -G -o libbar.so -lc -bexpall bar.o barfoo.o -bnoentry

libfoo.a:   foo.o foo.auto
            ld -G -o shr.o foo.o -bexpall -lc -bnoentry
            $(AR) $(ARFLAGS) $@ shr.o foo.auto

clean:      -rm -f *.o *.a *.so core main *.lst *.map
```

Figure 11. Autoloading archive members

The goal for the example in Figure 11 is to provide a module that will override a definition from another module. Looking at the rules for building `main`, we see that `libfoo.a` is listed before `libbar.so`. This latter module contains a definition for `bar()`, which calls `foo()`. The definition for `foo()` is in the same module, `libbar.so`.

When building `libfoo.a`, the shared module containing an alternate definition for `foo()` is in another shared module, which is then placed within an archive. The linker will not (even when using runtime linking) automatically create references to shared modules within archives. To force it to accept and reference a module within an archive, an autoload script is used (`foo.auto`).

This script tells the linker that the named module within the archive containing the script is to be referenced by the module being built. Therefore, when linking `main`, the linker examines `libfoo.a` first, sees the autoload script, and understands that a reference to the module `shr.o` within `libfoo.a` is to be placed in the loader section. The linker then examines `libbar.so`, sees the definition of the referenced symbol `bar()`, and creates a loader section reference to that file.

Since runtime linking is enabled in this example, the loader section will list modules in the order in which they were specified on the linker's command line. At runtime, the definition for `foo()` resolves all references from any module. This will then override the definition provided by the module `libbar.so`. Therefore, the correct output will be as follows:

```
> main
in bar()
in foo()
```

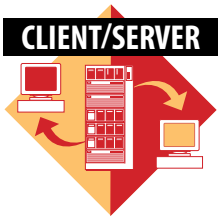
References

Online documentation is available at:
http://www.rs6000.ibm.com/doc_link/en_US/a_doc_lib/aixgen/

***Gary R. Hook**, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Hook joined IBM in 1990 and is currently an advisory software engineer in the IBM Server Group. He is the author of the Shared Libraries Hookable Symbols/6000 product and is currently a member of the AIX kernel development team, working on object file tools and the system loader. Mr. Hook has a BS in Electrical Engineering from the University of New Mexico.*

***Vandana Kumar**, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Ms. Kumar is an advisory engineer in the IBM Server Group. She has worked with several AIX application vendors in porting, enabling, and tuning their software on the RS/6000 platform. Ms. Kumar has a MS in Electrical Engineering from Wayne State University.*

***Stephen Peckham**, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Dr. Peckham is an advisory software engineer in the IBM Server Group. He developed the AIX Version 4 link editor and, for the past several years, he has been a member of the AIX kernel development team, working on the system loader and runtime linker. Dr. Peckham has a Ph.D. in Computer Science from Cornell University.*



IBM eNetwork Dispatcher and Oracle Net8

By Y. Joanna Wong

This article provides a brief overview about IBM eNetwork Dispatcher and outlines the steps to set up eNetwork Dispatcher to load-balance the Oracle® Net8™ connections from Oracle clients to Oracle8 Parallel Server running on either the IBM RS/6000 SP or an HACMP/6000 cluster of IBM RS/6000 workstations. Using IBM eNetwork Dispatcher, an Oracle client will connect to an Oracle Parallel Server(OPS) instance running on the least-loaded node without knowing specifically which instance.

The configuration described in this article was tested with IBM eNetwork Dispatcher Version 2.0 and Oracle8 Release 8.0.5 on the IBM RS/6000® SP™ running AIX® 4.2.1. Excerpts within this overview are drawn from the documentation on IBM eNetwork Dispatcher and Oracle8 Release 8.0.

IBM eNetwork Dispatcher

IBM eNetwork Dispatcher is scalable, load-balancing software that directs TCP/IP session requests to the appropriate TCP-based servers, based on a profile that can be customized. eNetwork Dispatcher consists of the Dispatcher function and the Interactive Session Support (ISS) function. The load on servers within a local area network or wide area network can be balanced using the Dispatcher function by itself, the ISS function by itself, or the Dispatcher function with the ISS function used as the monitor of the load on servers.

The Dispatcher load-balances requests at specific service levels, such as HTTP, FTP, SSL, NNTP, POP3, SMTP and Telnet. Client requests sent to the cluster address of the Dispatcher machine are directed to the server selected by the Dispatcher according to dynamically set weights and values. The selected server will then respond to the client requests directly. The list of servers for each service can differ among the services. The Dispatcher can be used by itself using rule-based load balancing (such as connections per second for a port) or can be used with a system-monitoring tool for load balancing.

The ISS function in eNetwork Dispatcher can monitor the load on servers for the Dispatcher. Figure 1 shows the Dispatcher with ISS as the monitor for load balancing. The ISS agents running on the servers periodically forward the level of activity to the ISS monitor. According to the metrics and policy chosen, such as the CPU load or the total number of users, the ISS monitor will broadcast the recommended IP address of the server for the ISS service to all ISS agents. The manager process of the Dispatcher function uses the input from ISS along with other criteria, such as the number of active connections and the number of new connections, to

Y. Joanna Wong

perform load balancing. Dispatcher will not forward client requests to servers that are detected to be unreachable.

The ISS function in eNetwork Dispatcher can balance the load on servers without the Dispatcher by using a domain name server (DNS). Figure 2 shows the use of ISS with domain name server for load balancing. The ISS monitor will broadcast the recommended IP address of the server for the ISS DNS name server to all ISS agents. The ISS agent running on the DNS will update the configuration files for the `named` daemon. Through name resolution, ISS will provide clients with the IP address of the least-busy server.

This article describes the steps to configure the ISS with a domain name server and Oracle Net8, and the Dispatcher with ISS (without a domain name server) and Oracle Net8 for load balancing. For the Dispatcher with ISS method, the database administrator must have root user permission to start the Oracle listeners on all the OPS instances following the steps outlined in the section "Configuring Oracle Net8 for Network Dispatcher Function" in this article. For the first method, once the ISS DNS name server is configured to run correctly, the database administrator can start the Oracle listeners as usual. For this reason, the first method is recommended if the database administrator is not the root user.

Configuring the ISS Function

The first step is to set up the domain name server on the ISS DNS name server. AIX 4.2.1 requires the boot file, the domain data file, the domain reverse data file, the domain local data file, and the domain cache file to correctly configure the ISS DNS name server. Specifics within the configuration files will vary depending on the AIX level and the setup of the site-wide domain name server. It is important to remember that the domain reverse data file for the ISS DNS name server should not have any network prefix. ISS will update the complete IP address of the ISS DNS service name in the domain data file and the domain reverse data file.

Sample configuration files provided in this article are for the configuration with the ISS DNS name server on

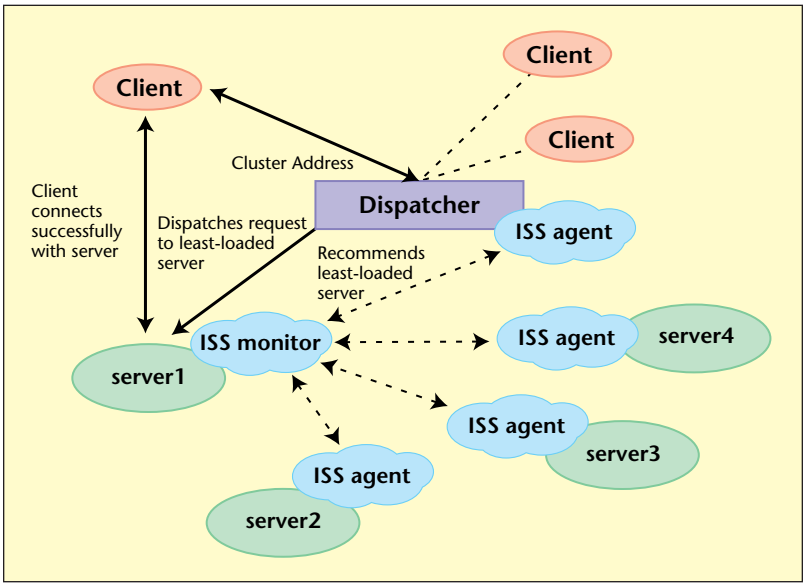


Figure 1. Dispatcher with ISS as the monitor for load balancing

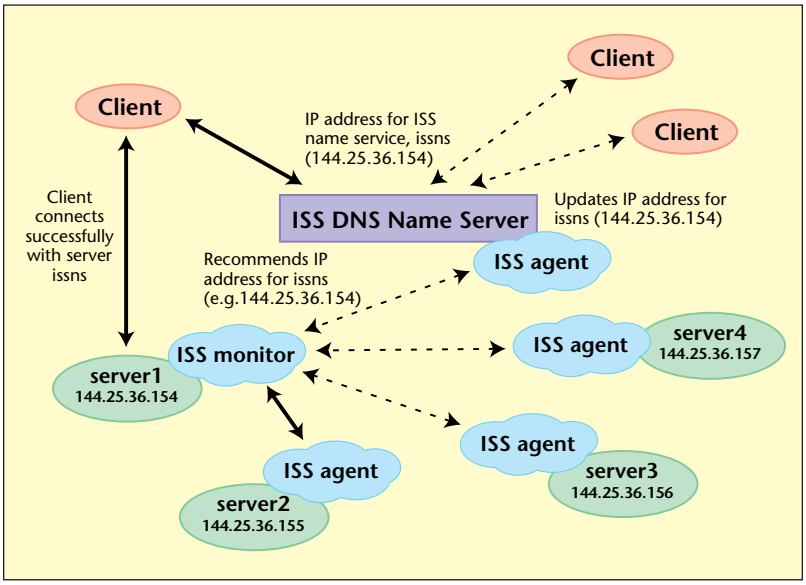


Figure 2. ISS with domain name server for load balancing

`server9.site.corp.com`. The type of Observer is `NameServer`. The ISS DNS name server will resolve requests for the subdomain `test.site.corp.com`.

The site-wide domain name server, `dns.site.corp.com`, is configured to delegate all resolution requests for the subdomain `test.site.corp.com` to the ISS DNS name server on `server9.site.corp.com`. The host name `issns.test.site.corp.com`

will be resolved to the IP address of the server with the minimum CPU load:

- server1.site.corp.com
- server2.site.corp.com
- server3.site.corp.com
- server4.site.corp.com

It is necessary to have root user permission to start and stop the named and the issd daemon. The ISS DNS name server can be started with /usr/bin/startsrc -s named -a "-b named.boot_file" where named.boot_file is the boot file for the ISS DNS name server. The named daemon can be refreshed after any change in the configuration files with /usr/bin/refresh -s named and can be stopped with /usr/bin/stopsrc -s named.

The ISS daemon is started with /usr/lpp/eND/iss/issd -c iss_config where iss_config is the configuration file. The ISS DNS name server configuration file is an ASCII file that has definitions for objects including the Cell, the Node, the Service, and the ResourceType. The iss_config configuration file should be the same on all nodes (server9, server1, server2, server3, and server4 in the example). The ISS daemon will operate as either the monitor or the agent depending on the parameters defined in the configuration file.

The Oracle client machines should be set up so that the ISS DNS name server (server9.site.corp.com) is the primary name server and the site-wide domain name server (dns.site.corp.com) is secondary; that is, the first entry in the file /etc/resolv.conf on the client machines is the hostname of the ISS DNS name server.

Sample Configuration Files for ISS

Four instances of the database opsdb with System Identifiers (SIDs) ops will run on four nodes: server1, server2, server3, and server4. To use ISS for load balancing, the SIDs of the parallel instances must be the same (For details, see the section in this article titled "Configuring Oracle Net8 for ISS"). Figure 3 shows the node addresses for the four nodes.

The ISS DNS name server is configured on server9.site.corp.com to resolve requests for the subdomain test.site.corp.com. The site-wide domain name server, dns.site.corp.com, is configured to delegate all resolution requests for the subdomain test.site.corp.com to the ISS DNS name server. Figure 4 shows the node name and IP address.

When the name server is started on the ISS DNS name server, server9.site.corp.com, the ISS service name, issns.test.site.corp.com, will be resolved to 144.25.36.154 (the IP address for server1) and 144.25.36.154 will be resolved to issns.test.site.corp.com. The IP address 144.25.36.154 in the domain data file (named.data) will be

Node Name	IP Address	SID
server1.site.corp.com	144.25.36.154	ops
server2.site.corp.com	144.25.36.155	ops
server3.site.corp.com	144.25.36.156	ops
server4.site.corp.com	144.25.36.157	ops

Figure 3. Four nodes for SIDs

Node Name	IP Address	Server Type
server9.site.corp.com	144.25.36.162	ISS DNS name server
dns.site.corp.com	144.25.88.144	Site-wide domain name server

Figure 4. Site-wide domain name server

updated to one of the following:

- 144.25.36.154
- 144.25.36.155
- 144.25.36.156
- 144.25.36.157

And 154.36.25.144 in the reverse data file (named.rev) will be updated periodically to one of the following:

- 154.36.25.144
- 155.36.25.144
- 156.36.25.144
- 157.36.25.144

ISS will periodically modify the IP address of the address resource record in named.data and the reverse address of the pointer resource record of issns.test.site.corp.com in named.rev to one of the IP addresses of server1, server2, server3, and server4. See Figures 5 and 6.

The forwarders line in the boot file specifies the address of the site-wide name server dns.site.corp.com that accepts recursive queries from other servers. The ISS DNS name server will send all queries that are not in its cache to the forwarders first. For AIX 4.3, forwarders is an option in the configuration file /etc/named.conf for the named daemon. See Figures 7, 8, and 9.

The Domain Data File (named.data)

```
;The SOA (Start of Authority) record

@    IN    SOA    server9.site.corp.com. root.server9.site.corp.com. (
                  1.1         ; Serial
                  3600        ; Refresh
                  300         ; Retry
                  3600000     ; Expire
                  86400 )     ; Minimum
;

; the NS (Name Server) records
;
test.site.corp.com.             IN    NS      server9.site.corp.com.
                                IN    NS      dns.site.corp.com.
dns.site.corp.com.              IN    A       144.25.88.144
;
; the A (address) records
;
; define local loopback host
;
loopback.test.site.corp.com.    IN    A       127.0.0.1
localhost.test.site.corp.com.   IN    CNAME   loopback.test.site.corp.com.
;
; define all hosts
;
issns.test.site.corp.com.       IN    A       144.25.36.154
;
```

Figure 5. Domain data file

The Domain Reverse Data File (named.rev)

```
; the SOA (Start of Authority) record  for the subdomain test.site.corp.com
;
test.site.corp.com IN SOA server9.site.corp.com. root.server9.site.corp.com.(
                              1.1         ; Serial
                              3600        ; Refresh
                              300         ; Retry
                              3600000     ; Expire
                              86400 )     ; Minimum
;
; the NS (Name Server) records
;
;  All name resolution that are not in the test.site.corp.com domain will be
;  forwarded to the site-wide domain name server dns.site.corp.com
;
                    IN     NS     dns.site.corp.com.
144.88.25.144       IN     PTR    dns.site.corp.com.
;
; the PTR (Pointer) records.     Reverse address mappings
;
154.36.25.144       IN     PTR    issns.test.site.corp.com.
;
```

Figure 6. Domain reverse data file

The Domain Local File (named.local)

```
;
; the SOA (Start of Authority) record
;
@    IN    SOA    server9.site.corp.com.  root.server9.site.corp.com. (
                              1.1         ; Serial
                              3600        ; Refresh
                              300         ; Retry
                              3600000     ; Expire
                              86400 )     ; Minimum
;
; the NS (Name Server) record
;
     IN    NS     server9.site.corp.com.
;
; the PTR (Pointer) records
;
1    IN    PTR    localhost.
;
```

Figure 7. Domain local file

The Domain Cache File (named.cache)

```
;
; the NS (Name Server) records
;
                         IN    NS    dns.site.corp.com.
;
; the A (Address) records
;
dns.site.corp.com.       IN    A     144.25.88.144
;
```

Figure 8. Domain cache file

The Boot File (named.boot_file)

```
domain      test.site.corp.com
primary     test.site.corp.com      /etc/named.data
primary     in-addr.arpa            /etc/named.rev
primary     0.0.127.in-addr.arpa    /etc/named.local
cache       .                       /etc/named.cache
forwarders  144.25.88.144
```

Figure 9. Boot file

ISS Configuration File for Update DNS Mode (iss_config)

Figure 10 shows a configuration file that illustrates how ISS can be used with a DNS name server to perform load balancing for a cluster of servers. Comments in the configuration file must begin with the # symbol. The minimum CPU load-balancing method is used for the cluster. Default values are used for parameters that are not explicitly declared.

All the keywords following each object (for example, `Cell`, `Node`) refers to the object until the declaration of the next object. For example, in Figure 10, keywords `LogLevel`, `HeartbeatInterval`, and `HeartbeatsPerUpdate` in the first stanza following the keyword `cell` refer to the local cell `isscell`.

Basic parameters for the whole cell. The sample configuration only has one local cell with the name `isscell` and one service with the name `service1`.

Data of individual nodes in the isscell cell. If the keyword `NotMonitor` refers to a `Node`, the `issd` process on this node cannot assume the role of an ISS monitor. It will run as an ISS agent or perform other special ISS functions. The node `server1` has ISS monitor priority 11 and will run the `issd` process in ISS monitor mode. The node `server2` has monitor priority 12 and will run as a backup ISS monitor when `server1` is down. The other nodes are declared `NotMonitor`.

The ISS agent on `server9`, where the DNS name server is running, will update the domain data file and the domain reverse data file.

Definition of CPU ResourceType. The specified `MetricLimits` indicates that a node will not be used if its CPU usage goes over 90% (fail limit) and will not be put back into the participation list until CPU usage drops below 80% (recover limit). ISS will not schedule work for nodes that are unreachable on the network or where the `issd` processes are not running. `CPULoad` and `FreeMem` are the two internal metrics available. Users can define their own external metrics.

Definition of services available in isscell cell. The only service configured in Figure 10 is `service1`, which has the DNS name `issns.test.site.corp.com`. The ISS observer is the `NameServer` in this configuration. ISS will work in conjunction with the DNS name server running on the node `server9.site.corp.com` to resolve the DNS name `issns.test.site.corp.com` to the IP address of one of the nodes associated with `service1`:

- `server1.site.corp.com`
- `server2.site.corp.com`
- `server3.site.corp.com`
- `server4.site.corp.com`

Since the `SelectionMethod` is `Best`, the name `issns.test.site.corp.com` will be mapped to one of the four IP addresses, `144.25.36.154`, `144.25.36.155`, `144.25.36.156`, or `144.25.36.157`—the one that has the minimum CPU usage. A new address is selected once every 60 seconds (`HeartbeatInterval*HeartbeatsPerUpdate`). Once every 20 seconds (`HeartbeatInterval`), ISS checks

that the server whose address is currently recommended is functional at its Internet Control Message Protocol (ICMP) layer. If at any time there are no servers available for `service1` and the name `issns.test.site.corp.com` is marked as `FAILED` in the DNS file, the alarm is triggered. In this example, an entry will be made in the file `/tmp/issalarm.log`.

```
# Basic parameters for the whole cell

Cell                    isscell     Local
LogLevel                Error
HeartbeatInterval       20
HeartbeatsPerUpdate     3


# Data of individual nodes in the isscell cell

Node            server9.site.corp.com    01
NotMonitor

Node            server1.site.corp.com    11

Node            server2.site.corp.com    12

Node            server3.site.corp.com    13
NotMonitor

Node            server4.site.corp.com    14
NotMonitor


# Definition of CPU ResourceType

ResourceType            CPU
Metric                  Internal    CPULoad
MetricWeight            1.0
MetricNormalization     0           100
MetricLimits            80.0        90.0
Policy                  Min


# Definition of Services available in isscell cell

Service             service1 issns.test.site.corp.com
NodeList            server1.site.corp.com
                    server2.site.corp.com
                    server3.site.corp.com
                    server4.site.corp.com
Alarm               echo "No available servers for service1">>/tmp/issalarm.log
ResourceList        CPU
SelectionMethod     Best


# The NameServer Observer running

Nameserver          server9.site.corp.com       53
ServiceList         service1
NamedData           /etc/named.data
NamedRev            /etc/named.rev
```

Figure 10. ISS configuration file for update DNS mode

The NameServer Observer running on the node server9.site.corp.com subscribes with service1. ISS will work with the DNS name server running on server9 to load-balance the cluster of servers (server1, server2, server3, and server4). The DNS name server daemon and the issd process running in monitor mode may reside on different nodes.

Configuring Oracle Net8 for ISS

In Oracle8, the network listener is the process set up to receive client connect requests on behalf of a service. The listener.ora file is the configuration file that is used when starting the listener. Each listener is configured to listen on the list of addresses given in the address_list. The address can be defined by the protocol, the host, and the port.

The Oracle System IDentifier (SID) of the database served by the listener is defined by the parameter sid_list_listener_name, where listener_name is the name of the listener. Oracle client initiates a connect request to the database server via Oracle Net8 using the service name. The service name in Oracle Net8 can be resolved with the Local Naming method. On the Oracle client, the local configuration file tnsnames.ora maps the service name to a network address and the Oracle SID given by the connect descriptor.

To configure Oracle Net8 with ISS for load balancing, the service name in the tnsnames.ora maps to the hostname of the ISS DNS service (issns.test.site.corp.com) and the Oracle SID of the database. The client initiates a connect request providing the service name.

The network address in the service name will be resolved by the ISS DNS name server to one of the IP addresses in the NodeList for the ISS DNS service. Oracle Net8 makes the connect request to this network address. The listener listening at the network address resolved by ISS receives the request and directs the request to the instance of the database it is serving. Since the client can connect to any of the servers for the ISS DNS service name using the same connect descriptor (same sid specified by connect_data in tnsnames.ora), the Oracle SID must be the same for all the OPS instances.

Several files in Oracle are uniquely distinguished by the Oracle SID of the database: the alert and trace files in $ORACLE_HOME/rdbms/log, the init.ora files, and the shared memory information in $ORACLE_HOME/dbs. If you are using an NFS-mounted Oracle home and using ISS with Oracle Net8 for load balancing, these files on all instances are no longer unique. The solution is to create soft links for the directories that cannot be shared—$ORACLE_HOME/dbs and $ORACLE_HOME/rdbms/log—to local directories. For example:

```
ln -sf /local/rdbms/log  $ORACLE_HOME/rdbms/log
ln -sf /local/dbs        $ORACLE_HOME/dbs
```

where /local/rdbms/log and /local/dbs are local directories on each of the database server nodes. Alternatively, the directory for the alert and trace files can be set in init.ora to be $ORACLE_HOME/dbs with the following:

```
background_dump_dest = ?/dbs
```

Then, $ORACLE_HOME/dbs will be the only directory that cannot be shared. The local init.ora file in /local/dbs on each node can be edited for instance-specific parameters, such as instance_number.

Sample Configuration Files for Oracle Net8 with ISS

Figures 11 and 12 represent sample configuration files for Oracle Net8 with ISS.

Oracle clients can connect to the least-loaded database instance with Oracle SID ops using the service name n805. For example:

```
sqlplus user/password@n805
```

Troubleshooting

Users may encounter the following error using ISS from IBM eNetwork Dispatcher Version 2.0.0.0. If the ISS monitor fails to recommend a server for the DNS name of the ISS service (for example, ISS agents are not started on any of the server nodes), the ISS agent on the ISS DNS name server will mark the failed entry in the domain data file (named.data):

listener.ora

```
listener=
(address_list=
  (address=
    (protocol=tcp)
    (host=server1.site.corp.com)
    (port=8006)
  )
)
sid_list_listener=
(sid_list=
  (sid_desc=
    (sid_name=ops)
    (envs='GMS_QUERY_FILE=/dev/rlv805.ogds,GMS_HC_SOCKET=/tmp/serv.gms,
           GMS_NODE_LIST=/805/gms/nodes.all,GMS_LOG_DIR=/tmp/805gms')
    (oracle_home=/805)
  )
)
wait_time_listener=0
connect_timeout_listener=10
trace_level_listener=off

listener2=
(address_list=
  (address=
    (protocol=tcp)
    (host=server2.site.corp.com)
    (port=8006)
  )
)
sid_list_listener2=
(sid_list=
  (sid_desc=
    (sid_name=ops)
    (envs='GMS_QUERY_FILE=/dev/rlv805.ogds,GMS_HC_SOCKET=/tmp/serv.gms,
           GMS_NODE_LIST=/805/gms/nodes.all,GMS_LOG_DIR=/tmp/805gms')
    (oracle_home=/805)
   )
 )
wait_time_listener2=0
connect_timeout_listener2=10
trace_level_listener2=off

listener3=
(address_list=
  (address=
    (protocol=tcp)
    (host=server3.site.corp.com)
    (port=8006)
  )
)
sid_list_listener3=
(sid_list=
  (sid_desc=
    (sid_name=ops)
    (envs='GMS_QUERY_FILE=/dev/rlv805.ogds,GMS_HC_SOCKET=/tmp/serv.gms,
           GMS_NODE_LIST=/805/gms/nodes.all,GMS_LOG_DIR=/tmp/805gms')
    (oracle_home=/805)
  )
```

(continued on following page)

Figure 11. listener.ora

(continued from previous page)

```
)
wait_time_listener3=0
connect_timeout_listener3=10
trace_level_listener3=off

listener4=
(address_list=
  (address=
    (protocol=tcp)
    (host=server4.site.corp.com)
    (port=8006)
   )
 )
sid_list_listener4=
(sid_list=
    (sid_desc=
       (sid_name=ops)
       (envs='GMS_QUERY_ FILE=/dev/rlv805.ogds,GMS_HC_SOCKET=/tmp/serv.gms,
              GMS_NODE_LIST=/805/gms/nodes.all,GMS_LOG_DIR=/tmp/805gms')
       (oracle_home=/805)
    )
 )
wait_time_listener4=0
connect_timeout_listener4=10
trace_level_listener4=off
```

Figure 11. listener.ora

tnsnames.ora:

```
n805=
(description=
  (address=
    (protocol=tcp)
    (host=issns.test.site.corp.com)
    (port=8006)
  )
  (connect_data=
    (sid=ops)
   )
 )
```

Figure 12. tnsnames.ora

```
issns.test.site.corp.com.FAILED
IN  A  144.25.36.154
```

and in the domain reverse data file (`named.rev`):

```
154.36.25.144      IN   PTR
issns.test.site.corp.com.FAILED
```

Hostname resolution for `issns.test.site.corp.com` will fail. When a new address is successfully selected at the next update (`HeartbeatInterval*HeartbeatsPerUpdate` seconds later), the failed entry will be updated to reflect the new IP address. Unfortunately, the update missed the vital dot (`.`) that appends the network address (`issns.test.site.corp.com`) in the domain data file and the domain reverse data file.

For example, in the domain data file:

```
issns.test.site.corp.com
IN    A    144.25.36.155
```

and in the domain reverse data file:

```
155.36.25.144    IN    PTR
issns.test.site.corp.com
```

Hostname resolution for `issns.test.site.corp.com` will continue to fail. To correct for the problem, follow these steps:

1. Stop the `issd` daemon on the node where the DNS name server is running: `kill -TERM issd_pid` where `issd_pid` is the PID of the `issd` process.
2. Edit the domain data file and the domain reverse data file to add the missing dot.
3. Update the `named` daemon with the new configuration files:

   ```
   /usr/bin/refresh -s named
   ```

4. Restart the `issd` daemon on the node where the DNS name server is running:

   ```
   /usr/lpp/eND/iss/issd -c iss_config
   ```

The vital dot will not be missed from future updates and hostname resolution for `issns.test.site.corp.com` will be successful.

To prevent encountering the missing-dot problem at startup, follow these steps:

1. Define the node where the ISS DNS name service is running (`server9`) to be `NotMonitor`.
2. Start `issd` on the node that is the primary ISS monitor (`server1`).
3. Start `issd` on the node that is the backup ISS monitor (`server2`).
4. Start `issd` on other nodes except the ISS DNS name server, where ISS agents will run (`server3`, `server4`).
5. Wait for one recommendation to occur (`HeartbeatInterval*HeartbeatsPerUpdate` seconds), then start `issd` on the ISS DNS name server (`server9`).

The missing dot will be corrected in future releases of IBM eNetwork Dispatcher.

For IBM eNetwork Dispatcher Version 2.0.0.1, the ISS agent on the ISS DNS name server will not update the entry if the ISS monitor fails to recommend a server for the ISS service because ISS is not running on any of the nodes. Hostname resolution for the ISS service will return the IP address that is in the domain data file (`named.data`) until the ISS monitor recommends a new IP address.

On the other hand, if all servers fail to meet the metrics (for example, on all servers, the CPU usage was over the fail limit but has not dropped below the recover limit), the ISS service will be marked as `FAILED` in the `named` configuration files. Users must follow the steps outlined above if they encounter the missing-dot problem.

For IBM eNetwork Dispatcher Version 2.0.0.0 and 2.0.0.1, if the ISS monitor runs on the same node as the ISS DNS name server, the ISS monitor fails to update the domain data file and the domain reverse data file. This will be changed in future releases of the IBM eNetwork Dispatcher. The sample configuration has the ISS monitor and the ISS DNS name server running on different nodes.

Configuring the eNetwork Dispatcher Function

The eNetwork Dispatcher function, unlike the ISS function, does not involve hostname resolution for load balancing. The cluster address is the hostname or IP address to which remote clients will connect. TCP services from remote clients to the cluster address will be directed by the Dispatcher to one of the servers configured for the cluster address. The different TCP services are identified by the specific port connections; for example, FTP is on port 20, and Telnet is on port 23. Each TCP server can be defined on a subset of all the ports configured for the cluster address.

The following will outline the steps to configure the eNetwork Dispatcher using the cluster address `tcpnd.site.corp.com` (`144.25.36.237`) on the Dispatcher

machine server9.site.corp.com (144.25.35.162). The TCP server machines are the database servers where parallel instances are started:

- server1.site.corp.com
- server2.site.corp.com
- server3.site.corp.com
- server4.site.corp.com

Oracle Net8 listeners are configured to listen on port 8006 of these servers. Root user permission is required to configure the eNetwork Dispatcher.

The non-forwarding address is the IP address used to connect to the Dispatcher machine for administrative purposes. By default, this is the IP address of the hostname. In this example, the non-forwarding address for server9.site.corp.com is 144.25.36.162, and it is defined on the standard network interface en0. To configure and start the Dispatcher, follow these steps:

1. On server9, create an alias on the network interface of the non-forwarding address, en0, to the cluster address 144.25.36.237 with the site-wide netmask, 255.255.252.0:

   ```
   /etc/ifconfig en0 alias
   144.25.36.237 netmask 255.255.252.0
   ```

   If successful, netstat -ni will show two IP addresses for the network interface en0.

2. On each of the TCP servers, server1, server2, server3, and server4, create an alias on the loopback device to the cluster address 144.25.36.237

   ```
   /etc/ifconfig lo0 alias
   144.25.36.237 netmask 255.255.252.0
   ```

   If netstat -nr returns any extra route to the cluster address, such as:

   ```
   Network     Netmask         Gateway         Interface
   144.0.0.0   255.255.252.0   144.25.36.237   144.25.36.237
   ```

   delete the extra route with the following:

   ```
   route delete -net 144.0.0.0
   144.25.36.237 netmask 255.255.252.0
   ```

3. The configuration file from /usr/lpp/eND/dispatcher/bin/samples/en_US/sample.cfg can be modified to configure and start the eNetwork Dispatcher on the Dispatcher machine, server9, as shown in Figure 13.

Shell Script to Configure eNetwork Dispatcher on the Dispatcher Machine

```
#!/bin/ksh
iam=`whoami`
if [ "$iam" != "root" ]
 then
  echo "You must login as root to run this script"
  exit 2
fi

echo "Starting the ND server.."
ndserver start
sleep 5

echo "Starting the executor.."
ndcontrol executor start

# Can use either hostname or IP address for the non-forwarding address, the
# cluster address, or the addresses for the TCP server machines
```

(continued on following page)

Figure 13. Configuring eNetwork Dispatcher on the Dispatcher Machine

(continued from previous page)

```
NFA=server9.site.corp.com
CLUSTER=tcpnd.site.corp.com
SERVER1=server1.site.corp.com
SERVER2=server2.site.corp.com
SERVER3=server3.site.corp.com
SERVER4=server4.site.corp.com

echo "Loading the non-forwarding address $NFA"
ndcontrol executor set nfa $NFA

echo "Loading CLUSTER address $CLUSTER"
ndcontrol cluster add $CLUSTER

# Services for port 8006 (Oracle Net8 listener)
#               port 23     (telnet)
# will be defined in this configuration

echo "Creating the ports for CLUSTER: $CLUSTER"
ndcontrol port add $CLUSTER:8006+23

# The telnet servers are server1, server2, and server3.
#  The Oracle Net8 listener servers are server1, server2, server3, and server4.

echo "Adding TPC server machines"
ndcontrol server add $CLUSTER:8006+23:$SERVER1
ndcontrol server add $CLUSTER:8006+23:$SERVER2
ndcontrol server add $CLUSTER:8006+23:$SERVER3
ndcontrol server add $CLUSTER:8006:$SERVER4

# The manager will use the following relative proportion of importance to find
# the combination that gives the best performance :
#      the number of active connections on each TCP server   = 10%
#      the number of new connections on each TCP server      = 10%
#      input from the advisor                                = 10%
#      input from the system monitoring tools, such as ISS   = 70%

echo "Starting the manager..."
ndcontrol manager start

ndcontrol proportions 10 10 10 70

# The advisors currently available with the software distribution are for the
# protocol  ftp, telnet, smtp, http, pop3, nntp and ssl.  Users can define
# their own advisors.

echo "Starting the Telnet advisor on port 23 ..."
ndcontrol advisor start telnet 23
```

Figure 13. Configuring eNetwork Dispatcher on the Dispatcher Machine

Configure the ISS as Monitor for eNetwork Dispatcher

The `issd` daemon is started on the TCP server machines and the Dispatcher machine with `/usr/lpp/eND/issd -c iss_config`, where `iss_config` is the ISS configuration file. The syntax of the ISS configuration file for the `Dispatcher` Observer is similar to that for the `Nameserver` Observer (see the section "Sample Configuration Files for ISS" at the beginning of this article).

The CPU load information is gathered by the ISS monitor and forwarded to the manager of the Dispatcher. The manager adjusts how the server machines are weighted on

each port and provides the new weighting to the executor to load-balance new client connections. The ISS monitor and the Dispatcher run on different machines in this configuration.

ISS Configuration File for Dispatcher Mode (iss_config)

Figure 14 shows a sample configuration file that illustrates how ISS can be used as a system monitoring tool with the Dispatcher to perform load balancing for a cluster of servers.

Nodes in isscell cell. The node `server1` has ISS monitor priority 11 and will run the `issd` process in ISS monitor mode. The node `server2` has monitor priority 12 and will run as the backup ISS monitor when `server1` is down. The other nodes are declared `NotMonitor`. The `issd` processes on these nodes will run as ISS agents.

Definition of CPU ResourceType. The value `CPULoad` should be minimized to identify the least-loaded node. When the value of the CPU on a node is more than 90%, the node is removed from active participation in the Service with which it is associated. The node is returned to active participation when the metric has a value of less than or equal to 80.

Definition of Services available in isscell cell. The service `dp_sqlnet` is intended to be used by a `Dispatcher` Observer. The cluster address, `tcpnd.site.corp.com`, and the port number 8006 for `dp_sqlnet` must be specified.

The type of observer in the `isscell` cell is `Dispatcher`. For this observer, the role of ISS is to gather information. The ISS monitor will evaluate the metrics associated with the service `dp_sqlnet` that it gathers from the ISS agents and forward the result to the Dispatcher that subscribes to this service every 60 seconds (`HeartBeatInterval*HeartbeatsPerUpdate`). The `Dispatcher` will use this information, along with other sources of information, to perform load balancing.

```
# Basic parameters of the whole cell

Cell                    isscell     Local
LogLevel                Error
HeartbeatInterval       20
HeartbeatsPerUpdate     3



# Nodes in isscell

Node         server9.site.corp.com     01
NotMonitor

Node         server1.site.corp.com     11

Node         server2.site.corp.com     12

Node         server3.site.corp.com     13
NotMonitor

Node         server4.site.corp.com     14
NotMonitor



# Definition of CPU ResourceType

ResourceType            CPU
Metric                  Internal       CPULoad
Policy                  Min
MetricWeight            1.0
MetricNormalization     0.0            100.0
MetricLimits            80.0           90.0



# Definition of Services available in isscell

Service           dp_sqlnet  tcpnd.site.corp.com  8006 0
SelectionMethod   Best
ResourceList      CPU
NodeList          server1.site.corp.com
                  server2.site.corp.com
                  server3.site.corp.com
                  server4.site.corp.com



# The type of the Observer in the isscell cell is
# Dispatcher.

Dispatcher        server9.site.corp.com    10004
ServiceList       dp_sqlnet
```

Figure 14. ISS configuration file for Dispatcher Mode

The Dispatcher and the ISS monitor may run on the same node. In this configuration, they are on different nodes: Dispatcher is on `server9` and the ISS monitor is on `server1`.

Configuring Oracle Net8 for eNetwork Dispatcher Function

To configure Oracle Net8 with eNetwork Dispatcher for load balancing, the service name in the `tnsnames.ora` file maps to the host name of cluster address (`tcpnd.site.corp.com`), the TCP port, and the Oracle SID of the database. The client initiates a connect request providing the service name. The connect request will be forwarded by the Dispatcher to one of the TCP servers configured to the port of the cluster address. The listener listening at the port of the cluster address on the selected server accepts the client request, directs the request to the OPS instance it is serving, and establishes a connection with the client without involving the Dispatcher machine.

The listeners for the OPS instances are defined by its address (`protocol`, `host`, `port`). To use eNetwork Dispatcher for load balancing, the listeners on the different server nodes are defined to listen at the same port of the cluster address and have the same configuration. When the process `tnslsnr` starts the listener, it will succeed only if `tnslsnr` first fails to connect to the listener on the network. Once a listener is started on an OPS node, attempts to start listeners with the same configuration on the other OPS nodes will fail because `tnslsnr` can connect to the listener that has already started. To use eNetwork Dispatcher with Oracle Net8, the difficulty is to start listeners with the same configuration on more than one server.

As a root user, one can stop TCP service from being dispatched to a server temporarily with `/usr/bin/ndcontrol manager quiesce server_host` where `server_host` is the IP address or the hostname of a TCP server for the cluster address. The manager will set the weight to be 0 for this server in every port where it is defined. No more connections will be sent to this server. If this command is executed for all servers except one, the listener can be started successfully only on this one server:

```
$ORACLE_HOME/bin/lsnrctl start listener
```

The Dispatcher can resume dispatching TCP services to the `server_host` with `/usr/bin/ndcontrol manager unquiesce server_host`.

Listeners with the same configuration file can be started on more than one TCP server by repeating the above sequence of steps as shown in Figure 15.

Dispatcher Machine	**TCP Servers**
Dispatcher machine (`server9`), root user permission required Stop dispatching TCP services to all servers (for example, `server1`) `ndcontrol manager quiesce server1`	Each TCP server (`server1`, `server2`,...)

For each TCP server (such as `server1`)
Resume dispatch to server:
`ndcontrol manager unquiesce server1`
 	
Stop dispatch to server (`server1`):
`ndcontrol manager quiesce server1`
	Start listener on server (`server1`):
	`lsnrctl start listener`

After starting listener on all servers, resume dispatch to all servers (for example, `server1`): `ndcontrol manager unquiesce server1`	

Figure 15. Starting listeners on multiple servers

When using eNetwork Dispatcher with Oracle Net8, the log files in `$ORACLE_HOME/rdbms/log` are no longer unique because the listener names are the same on all OPS database server nodes. If using NFS-mounted Oracle home, the directory `$ORACLE_HOME/network/log`, in addition to `$ORACLE_HOME/rdbms/log` and `$ORACLE_HOME/dbs`, cannot be shared (see discussion in "Configuring Oracle Net8 for ISS"). You can create a soft link to a local directory for `$ORACE_HOME/network/log` as follows:

```
ln -sf /local/network/log
$ORACE_HOME/network/log
```

where `/local/network/log` is a local directory on each of the servers to the cluster address. Alternatively, the `trace_directory_listener` and `log_directory_listener` can be defined in the configuration file `listener.ora` to be a local directory (such as `/local/network/log`), instead of the default directory `$ORACLE_HOME/network/log`.

Sample Configuration Files for Oracle Net8 with eNetwork Dispatcher

Figures 16 and 17 show sample configuration files for Oracle Net8 with eNetwork Dispatcher.

listener.ora :

```
listener=
(address_list=
  (address=
    (protocol=tcp)
     (host=tcpnd.site.corp.com)
     (port=8006)
   )
 )
sid_list_listener=
(sid_list=
  (sid_desc=
    (sid_name=ops)
    (envs='GMS_QUERY_FILE=/dev/rlv805.ogds,GMS_HC_SOCKET=/tmp/serv.gms,
           GMS_NODE_LIST=/805/gms/nodes.all,GMS_LOG_DIR=/tmp/805gms')
    (oracle_home=/805)
   )
)
trace_directory_listener=/local/network/log
log_directory_listener=/local/network/log
wait_time_listener=0
connect_timeout_listener=10
trace_level_listener=off
```

Figure 16. listener.ora

tnsnames.ora:

```
n805=
(description=
  (address=
    (protocol=tcp)
    (host=tcpnd.site.corp.com)
    (port=8006)
   )
  (connect_data=
    (sid=ops)
  )
 )
```

Figure 17. tnsnames.ora

Oracle client connect request to `n805` (for example, `sqlplus user/password@n805`) will be dispatched to the least-loaded OPS instance with Oracle SID `ops`.

References

For IBM eNetwork Dispatcher:

IBM eNetwork Dispatcher User's Guide Version 2.0 (GC31-8496-02)

Load-Balancing Internet Servers (an IBM Redbook) (SG24-4993-00)

URL for IBM eNetwork Dispatcher: *http://www.software.ibm.com/enetwork/dispatcher*

For Oracle Net8:

Oracle Net8 Administrators Guide, Release 8.0 (A58230-01)

***Y. Joanna Wong**, IBM Corporation, 1055 Joaquin Road, Mountain View, CA 94043. E-mail: yjw@us.ibm.com. Dr. Wong is a senior engineer/scientist at IBM and is currently a technical consultant at Oracle Corporation. She has worked on parallel programming of scientific and technical applications on the IBM RS/6000 SP and has focused on performance benchmarks for customers. Dr. Wong holds a PhD and MS in Theoretical Physics from Cornell University and an AB in Physics from Princeton University.*

Of JavaBeans, RMI, and VisualAge for Java

By Greg Flurry

JavaBeans, which provide the component model for Java-based applications, are specifically intended for use with visual builder tools like VisualAge for Java from IBM. The Java Remote Method Invocation (RMI) mechanism allows the distribution of Java applications across machine boundaries. Unfortunately, distributing JavaBeans-based applications with RMI is not easy. This article examines the issues and shows how the RMI Access Builder in VisualAge for Java can provide solutions.

JavaBeans™ (or just beans) enable Java™ programmers to define reusable, modular components. In addition, JavaBeans provide mechanisms that allow these components to be easily manipulated in visual builder tools. This article provides a brief introduction to beans; many references[1,2,3] can provide more in-depth information.

A bean generally follows a particular design pattern based around properties, methods, and events. In fact, a Java class does not need to extend any `Bean` class for it to be considered a bean.

A *property* is part of the bean's internal state, typically a variable defined in the bean's class. The bean design pattern dictates that properties be accessed by public methods for getting and setting the properties. The *methods* for a bean are those public methods not related to properties.

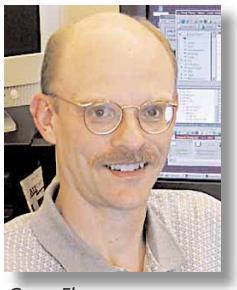
Events are the interesting part of beans. JavaBeans can define and generate their own events using the Java 1.1 event model. The bean design pattern dictates that a bean must define the following to generate an event:

- The event type (class)
- An *event listener* interface that accepts events of that type
- Methods to add and remove listeners for that particular event type

The bean design pattern also dictates that a class wanting to receive events of that type must implement the event listener interface and call a method to add itself to the generating bean's list of listeners. It may also call a method to remove itself from the list.

A JavaBeans Example

We will examine a simple chat application to demonstrate the various aspects of



Greg Flurry

[1]Flurry, Greg. "VisualAge for Java on AIX." *AIXpert*. March 1999.
[2]Brookshier, Dan. *JavaBeans Developer's Reference*. Indiana: New Riders Publishing. 1997.
[3]Sridharan, Prashant. *JavaBeans Developer's Resource*. New Jersey: Prentice-Hall. 1997.

JavaBeans. The concepts behind this example are derived from the chat application described in "Applets and Servlets: A Smooth Blend" by Greg Flurry and Jeff Jilg in the June 1998 issue of *AIXpert*. The example for this article will implement the server as a bean rather than a servlet. We will build this example using IBM's VisualAge® for Java, Enterprise Edition 2.0. This tool runs on AIX® and other platforms.

VisualAge for Java (VAJ) is a good environment for quickly building and testing Java applications. We assume readers are familiar with VAJ, so we will concentrate on the tasks to be done rather than how to do them.

Figure 1 shows the basic design of the application. The client consists of two threads:

- The GUI thread presents a user interface that allows messages to be sent and viewed.
- The Listener thread simply waits for incoming events from the server and displays them.

Using two threads eliminates potential deadlock conditions in the client that result from listening for events from the GUI thread and the chat server. The server bean has a method that allows a client to post a message to the server; the server then sends events to all listeners. The server bean is initiated as a single thread in this example, which limits it to a single client. If the server deals with multiple clients, it would need to be more complex. We will see later why this is not necessary.

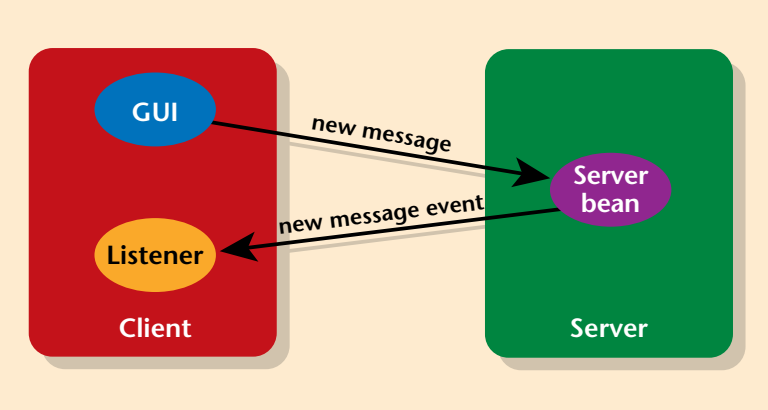


Figure 1. Chat application design

We first define the event that the server bean will generate. Figure 2 shows the ChatMessageEvent. As with all bean events, ChatMessageEvent extends

```
public class ChatMessageEvent extends java.util.EventObject {
   private String name;
   private String text;

public ChatMessageEvent(Object obj, String name, String text) {
   super(obj);
   this.name = name;
   this.text = text;
}

void setName(String newValue) {
   this.name = newValue;
}
public String getName() {
   return name;
}

void setText(String newValue) {
   this.text = newValue;
}
public String getText() {
   return text;
}
}
```

Figure 2. The ChatMessageEvent

java.util.EventObject. ChatMessageEvent has two properties and the corresponding methods to set and get them. The name property contains the name of the user sending a message and the text property contains the message.

Figure 3 shows the ChatMessageEvent listener interface. As with all event listener interfaces, ChatEventListener extends java.util.EventListener. ChatEventListener defines a single method that delivers the event to the listener.

Next is the server bean. Figure 4 shows the ChatServer bean, which includes the methods required to add and remove a listener: addChatListener() and removeChatListener(). The newMessage() method allows chat clients to give the server a message to be distributed to all clients. The setAlias() method, a special form of sending a message to the server, indicates that a new client has joined a chat session. Similarly, unsetAlias() indicates that a client has left a session.

The following concessions that affect the eventual use of the server bean by multiple clients are worth noting:

- The Vector chatListeners holding the list of clients is declared static so that only one copy exists for the class.
- The methods that add and remove listeners are synchronized on chatListeners to keep the list correctly linked.
- The method that sends events to clients clones chatListeners before using it for the same reason.
- The newMessage() method is synchronized.

This chat server differs from the servlet in the referenced article. The servlet sends a salutation message when a client joins or leaves only to that client. The servlet can do this because each client had its own server thread that remembers the alias for a client.

This is not possible for the chat server in this article because of the restrictions addressed below. Instead, this ChatServer sends a salutation to all clients currently joined in the chat session; see the setAlias() and unsetAlias() methods in Figure 4. This actually could be a better solution because all clients know when a client joins or leaves.

Figure 5 shows the ChatListener class that implements client-side listener for events from the chat server. Notice that it implements the ChatEventListener interface as well as a setTarget() method that identifies a TextArea in the user interface thread of the client used to display the properties of the received event.

```
public interface ChatEventListener extends java.util.EventListener {

    void newChatMessage(ChatMessageEvent m);
}
```

Figure 3. The ChatEventListener Interface

```
public class ChatServer {
   static Vector chatListeners = null;

public ChatServer() {
   super();
   if (chatListeners == null) {
      chatListeners = new Vector();
   }
}

public void addChatListener(ChatEventListener listener) {
   // add listener to list
   synchronized (chatListeners) {
```

(continued on following page)

Figure 4. The bean ChatServer

(continued from previous page)

```
      chatListeners.addElement(listener);
   }
}

public void removeChatListener(ChatEventListener listener) {
   // remove listener from list
   synchronized (chatListeners) {
      chatListeners.removeElement(listener);
   }
}

public synchronized void newMessage(String name, String text) {
   //create a new event
   ChatMessageEvent e = new ChatMessageEvent(ChatServer.this,
      name, text);
   // fire off the event
   Vector list = (Vector) chatListeners.clone();
   for (int i=0; i<list.size(); i++) {
      ChatEventListener l = (ChatEventListener) list.elementAt(i);
      l.newChatMessage(e);
   }
}

public void setAlias(String alias) {
   // send event to welcome listener

   newMessage("Chat Server", alias+" has joined; welcome!");
}

public void unsetAlias(String alias) {
   // send event to remove listener
   newMessage("Chat Server", alias+" has left; goodbye!");
}
}
```

Figure 4. The bean ChatServer

```
public class ChatListener extends Thread implements ChatEventListener
{
   private java.awt.TextArea outTarget;

public ChatListener() {
   super();
}
/** Other VisualAge for Java generated constructors deleted
  */

public void newChatMessage(ChatMessageEvent m) {
   outTarget.append(m.getName() + ": " + m.getText() + "\n");
}

public void setTarget(java.awt.TextArea target) {
   outTarget = target;
}
}
```

Figure 5. The ChatListener class

We used the VAJ Visual Composition Editor to create the `ChatClient` class that presents the user interface for the chat client. Figure 6 shows the Visual Composition Editor free-form surface with the completed chat client. The **Connect** button brings up a dialog that allows the user to enter an alias. The **OK** button on the dialog invokes the `ChatClient.connectMe()` method (see Figure 7).

The **Send** button invokes the `newMessage()` method on the chat server (ChatServer1). The **Disconnect** button invokes the `ChatClient.disconnectMe()` method (see Figure 7).

Figure 7 shows the hand-coded parts of the `ChatClient` class. The fields and methods related to the user interface and the various connections between beans on the free-form surface are not shown, because they are generated automatically by the VAJ Visual Composition Editor. The `ChatClient.connectMe()` method requires the alias, the chat listener, and the chat server as parameters. The method first calls the server to add the listener, and then calls the server again to send the alias for the client. The server generates a "welcome" event in response to the alias.

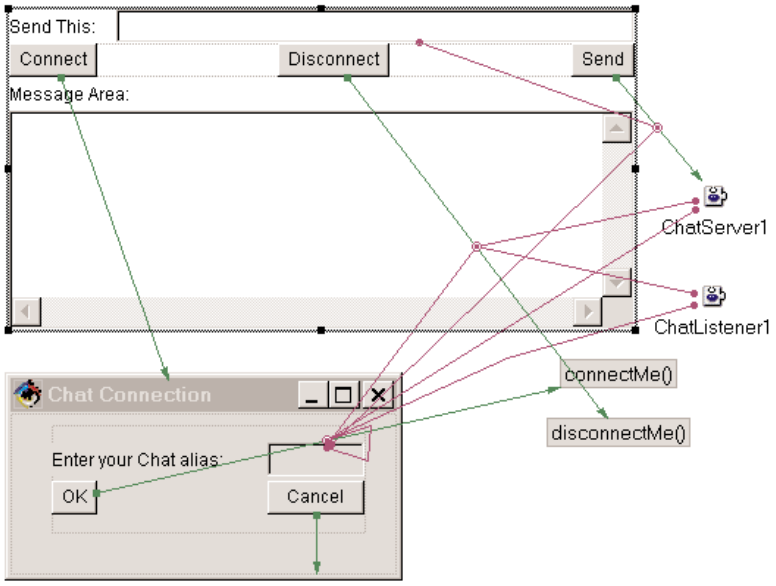


Figure 6. The ChatClient on the VAJ Visual Composition Editor free-form surface

The `ChatClient.disconnectMe()` method also requires the alias, the chat listener, and the chat server as parameters.

```
public class ChatClient extends Applet implements
    java.awt.event.ActionListener {
   private boolean connected = false;

private void connectMe(String alias, ChatListener listener,
    ChatServer server) {
    // register the listener with the server
    server.addChatListener(listener);
    // get rid of the dialog
    ivjDialog1.dispose();
    // set alias in the server
    server.setAlias(alias);
    // indicate connected
    connected = true;
}

private void disconnectMe(String alias, ChatListener listener,
    ChatServer server) {
    // unset alias in the server
    server.unsetAlias(alias);
    // remove the listener from the server
    server.removeChatListener(listener);
    // indicate not connected
    connected = false;
}
```

Figure 7. The hand-coded parts of the ChatClient class

The method first calls the server to send the alias for the client; the server generates a "goodbye" event in response to the alias. The method then calls the server again to remove the listener.

Figure 8 shows the result of running the ChatClient and performing the following actions:

1. Pressing the **Connect** button, which brings up the Connection dialog (not shown)
2. Entering "Greg" into the **Alias** text-entry field
3. Hitting the **Go** button in the dialog
4. Entering "This is the one and only client" into the **Send This:** text-entry field
5. Pressing the **Send** button
6. Hitting the **Disconnect** button

Although multiple ChatClients can run simultaneously, each will act as if only one client is running. That is because each time a ChatClient runs, VAJ starts a new Java Virtual Machine; therefore, each ChatClient gets its own instance of the `ChatServer` class. We could change the design to start multiple clients per server, but Java RMI and VAJ RMI Access Builder will eliminate the need for this.

Figure 8. The ChatClient in action

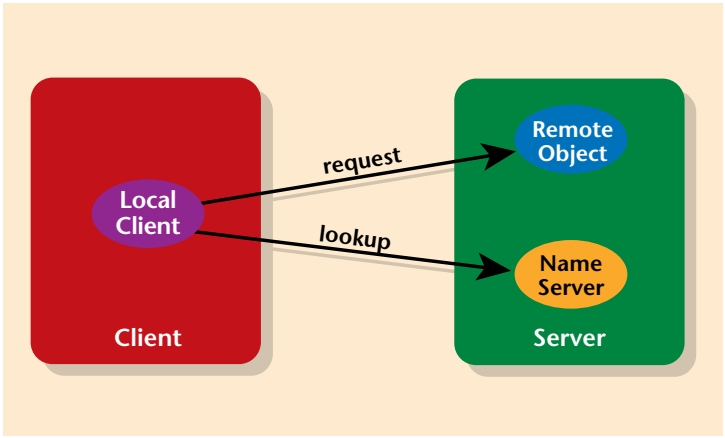


Figure 9. RMI operation

Remote Method Invocation

The Java Remote Method Invocation (RMI) provides Java programmers with mechanisms to create distributed Java applications. In simple terms, RMI allows a client object to call a remote server object in another Java Virtual Machine, whether that remote object resides on the same physical machine or on a physical machine in another country.

Figure 9 shows a high-level view of RMI operation. The local client uses a name server to get a remote object reference so the local object can access the remote object. The remote object must be running in a remote object server and have previously registered an instance of the remote object with the name server. After getting the remote object reference, the client object can call the remote object as if it were present locally.

The actual implementation and deployment of a remote object and a client require the definition of remote interfaces, the creation of stubs and skeletons for the remote objects, and other considerations. The article "Distributed Computing with Java Remote Method Invocation," by Greg Flurry in the July 1999 issue of *AIXpert* describes RMI and provides an example.

JavaBeans and RMI

RMI makes it easy to create distributed Java applications. Unfortunately, RMI does not support JavaBeans very well. Consider the chat example above. In this application, we would like to distribute the chat server bean remotely from clients. We also would like the clients to invoke methods on the server bean.

RMI handles this requirement; we would simply follow the practices outlined in the

article referenced above to make the chat server bean a remote object. We also want the chat server bean to send clients events, which means the server invokes a method on a client-side object implementing a listener interface. This requires that the client-side object also become a remote object.

In general, to create a distributed application with JavaBeans, a server bean must be a remote object from the client's perspective. To support events, the client must act as a remote object from the server's perspective. This can make creating applications with distributed beans difficult and error-prone.

RMI Access Builder

Fortunately, VisualAge for Java, Enterprise Edition 2.0 offers a solution to the JavaBeans/RMI problem. The RMI Access Builder can take existing beans and automatically generate the distribution-specific interfaces and classes to support method calls on a remote bean and the delivery of events to clients of the remote bean.

This article briefly describes the RMI Access Builder and how to use it. Detailed information is available in the soft-copy booklet *RMI Access Builder* found in the PDF documentation directory; for AIX, the file name for the booklet is */usr/IBMVJava/pdf/en_US/rmibld.pdf*.

The VAJ Help Home Page also provides information about the RMI Access Builder VAJ through the **Help** menu on the VAJ menu bar. Follow the links for Concepts, Access to the Enterprise, then RMI Access Builder. You can find additional information by initiating a search for "RMI" from the VAJ Help Home Page.

The RMI Access Builder (or simply Builder) has three components: Create Proxy Bean SmartGuide, Remote Object Instance Manager, and runtime class library. The Create Proxy Bean SmartGuide generates all the interfaces and classes necessary to use a bean remotely. In particular, it generates *proxy beans*. Client-side proxy beans are used to build clients of the remote bean; the proxy bean resides on the client machine, but acts like the remote bean. The server-side proxy bean works in conjunction with the remote bean to help deliver events and exceptions to clients through the client-side proxy bean.

The Remote Object Instance Manager provides an environment for running instances of remote beans and the corresponding server-side proxy beans. It is used both during development within VAJ and during deployment of remote beans developed within VAJ.

The runtime class library provides runtime services for both the client and the server; you should be concerned with it only during deployment of an application built with the Builder.

Installing RMI Access Builder

The RMI Access Builder is not installed by default when you install VAJ. To install the Builder, bring up the **Quick Start** menu, select **Features**, select **Add Features**, hit **OK**, then select **IBM Enterprise RMI Access Builder Library 2.0** and hit **OK**. The project IBM RMI Examples 2.0 will also be installed. The appropriate filesets must be installed during system installation before you can add the Builder feature or access the documentation and help pages.[4]

The Bean Example Revisited

We will now demonstrate how to use the RMI Access Builder to distribute the chat application from the example above. The first step is generating the proxy beans and other interfaces and classes using the Create Proxy Bean SmartGuide. Follow these steps:

1. In the VAJ Workbench, select the `ChatServer` class.
2. Bring up the Selected menu, select **Tools** to bring up the Tools submenu.
3. In the Tools submenu, select **Remote Bean Access** to bring up the Remote Bean Access menu.
4. In the Remote Bean Access menu, select **Create Proxy Beans...** to bring up the Create Proxy Bean SmartGuide.

[4]For details, see "VisualAge for Java on AIX " by Greg Flurry in *AIXpert*, March 1999.

5. In the Create Proxy Bean SmartGuide, enter a Proxy Bean Name. The default for the example will be ChatServer1, and it will be used for the rest of the example. The SmartGuide will already have the fields for bean class, project, and package filled out correctly.
6. In the Create Proxy Bean SmartGuide, select the checkbox for **Create RMI stub and skeleton for generated classes**. If you do not do it now, you will have to do it later, and there is no reason to wait.
7. In the Create Proxy Bean SmartGuide, select the checkbox for **Instantiate server object**. This will start the Remote Object Instance Manager and the name server.
8. In the Create Proxy Bean SmartGuide, hit **Finish**.

The Create Proxy Bean SmartGuide generates the client-side and server-side proxy beans. It then starts a name server (in this case, the RMI Registry) and an instance of the remote ChatServer bean in the Remote Object Instance Manager.

The Remote Object Instance Manager presents a user interface through which you can manage remote beans. You can start and stop bean instances. You can also request that each method call to a remote bean be shown in the log window of the user interface, which is a convenient tool for debugging.

The second step in distributing the chat application is to modify the ChatClient to use the client-side proxy for the remote ChatServer bean instead of the ChatServer bean itself. Follow these steps:

1. Import `COM.ibm.ivj.eab.rmi.client.*` in the `ChatClient` class declaration; this permits the understanding of remote exceptions.
2. Open the ChatClient to the Visual Composition Editor.
3. Add the client-side proxy to the free form surface; for this example the bean class to add is `chat.ChatServer1`. You can enter a name, but it will accept the default name ChatServer11.
4. Open the properties pop-up for ChatServer11 and change the parent component property to "this".
5. Modify the `ChatClient.connectMe()` and `ChatClient.disconnectMe()` methods so that the third parameter type is ChatServer1 instead of ChatServer; this will generate some errors that will be corrected in the next step.
6. Move all the connections to local ChatServer1 bean to the proxy ChatServer11 bean; this will correct the errors generated in the previous step.
7. Delete the local chat server bean (ChatServer1) from the free-form surface.
8. Save the modified ChatClient.

Now you can run the ChatClient as either an applet or an application in the VAJ integrated development environment (IDE). By following the steps described above, you should duplicate the results shown in Figure 8. You will also notice some slight delays in response due to the remote nature of the ChatServer. Now you can also start a second client and really do some chatting. To demonstrate multiple clients, do the following:

1. Start two instances of ChatClient, either as applets or applications
2. From instance 1, connect to the server with the alias "Greg"
3. From instance 2, connect to the server with the alias "Flurry"
4. From instance 1, send "Hello from instance 1"
5. From instance 2, send "Hello from instance 2"
6. From instance 1, disconnect
7. From instance 2, disconnect

Figure 10 shows the results you will see for instance 1 of the ChatClient. Figure 11 shows the results you will see for instance 2 of the ChatClient.

More About RMI Access Builder

RMI Access Builder makes it easy to create a distributed application with JavaBeans. Although you must understand some design considerations and limitations, many are the same as those for any RMI-based application, such as those dealing with the passing of objects as parameters. Others are unique to the RMI Access Builder environment. The most important of these considerations include:

- All exceptions but those thrown by the user are converted to `COM.ibm.ivj.eab.rmi.client.IVJRException`.
- The Remote Object Instance Manager must be restarted when the public interface of the remote bean is changed, and generally should be restarted when the implementation of the remote bean is changed.
- You must ensure that the properties (such as the parent component) of the client-side proxy are set correctly; the property settings depend on whether the client is run as an applet or a standalone application.

Some limitations on distributed applications when using the Builder include the following:

- The remote bean must support a default constructor, because server objects are instantiated using that constructor.
- It is possible to create deadlocks when a client waits for interactive events, such as button pushes in an event listener method. This is one reason to have the chat client multithreaded.
- User-defined event listener method signatures must be unique across all event listener interfaces used by the server bean.
- User-defined bean event types must exhibit the following characteristics. The event type cannot be a final class. The event type must support a constructor with the first parameter of type `java.lang.Object`; that constructor must have the lowest number of parameters of all the constructors. All other parameters in that constructor can only have Java default types.
- Remote server beans cannot produce `java.awt.event.*` event types.

Figure 10. Instance 1 of ChatClient with a remote ChatServer

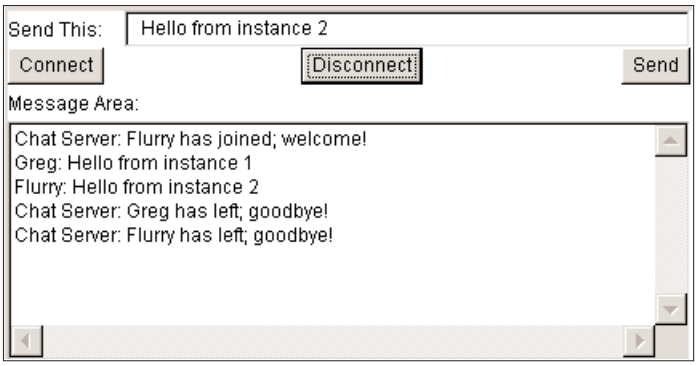


Figure 11. Instance 2 of ChatClient with a remote ChatServer

The *RMI Access Builder* booklet or the VAJ Help pages can provide more details on design considerations and restrictions.

During the design and testing of the chat application described above, we discovered some additional apparent limitations on RMI Access Builder that cannot be verified from the documentation, but seem worth mentioning. They may save some time as you build your own distributed applications with the Builder:

- The remote bean methods to add and remove event listeners MUST have the signatures:

```
void add<listener_name>(listener_interface_type)
void remove<listener_name>(listener_interface_type)
```

If such methods are not defined, the Create Proxy Bean tool will not create the

correct interfaces. It is possible to define add and remove methods with other signatures (for example, with additional parameters), but using them causes serialization exceptions.

- It is not possible to use the `listener_interface_type` as a parameter for any other method in the remote bean interface; doing so causes serialization exceptions.
- It is not possible for the remote bean to successfully generate an event in the add/remove methods; the methods complete successfully, but the client misses the generated events. It is possible to generate events in other methods of the remote bean without problems.

These additional limitations are partially responsible for the fact that the chat client registers with the server via an add method, then sends the alias with another method rather than have an add method with the alias as a parameter.

After developing and testing your distributed application, you will obviously want to deploy it. This takes place in two phases: one for the server and one for the client. For the server phase, you must package and export the remote bean, the server-side classes and interfaces generated by the Create Proxy Bean tool, the Builder runtime library, and the Remote Object Instance Manager. If the client will be deployed as an applet, you must also ensure that the runtime class library is available to the Web server so it can be accessed by the applet.

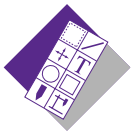
The deployment process for the client depends on whether you are deploying as an applet or application. For an applet, you need to package and deploy the client-side proxy bean and related interfaces and classes on the server in the same directory as the server-side code. You also need to create and deploy an HTML file to reference the applet; you must also ensure that the Web browser running the applet has compatible RMI support. In addition, there may be some firewall considerations.

For an application, you need to package and deploy the client-side proxy bean and related interfaces and classes and the Builder runtime class library on the client. For more details on deployment, see either the *RMI Access Builder* booklet or the VAJ Help pages.

VisualAge for Java Strikes Again

The RMI Access Builder from IBM's VisualAge for Java, Enterprise Edition 2.0 makes building distributed applications based on JavaBeans and RMI easy, and the restrictions placed on an application are not onerous.

Since the application can be developed and tested completely within VAJ, you can take advantage of all of VAJ's IDE features, such as the Visual Composition Editor, the debugging tool, and others. The VAJ RMI Access Builder boosts programmer productivity for Java-based, cross-platform distributed applications.



***Greg Flurry**, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Flurry is a senior technical staff member in the Software Solutions Division. His responsibilities, as part of the Software Group Technical Center, include developing and fostering IBM's e-Business technologies. He has a BS in Electrical Engineering from Vanderbilt University and an MS in the same field from the University of Kentucky.*

Lawson Software and Informix: High Availability Using IBM HACMP for AIX

By James T. West

Lawson Software, Informix, and IBM have teamed up to apply the benefits of IBM High Availability Cluster MultiProcessing (HACMP) software to LAWSON INSIGHT Business Management System and Informix Dynamic Server on an RS/6000 HA50 running the IBM AIX operating system. This provides users with a proven high-availability system to meet mission-critical needs for 24x7 availability for the most demanding server needs where downtime is disastrous.

In today's global economic climate, 24x7 availability of enterprise data is becoming essential to normal operations of many businesses. Downtime can have disastrous financial consequences.

Lawson Software™, Informix®, and IBM® have teamed up to apply the benefits of IBM High Availability Cluster MultiProcessing (HACMP) software to LAWSON INSIGHT® Business Management System and Informix Dynamic Server™ on an RS/6000® HA50 running the IBM AIX® operating system. End users of such a system will find the server and software to be more immediately available, even after dramatic, unplanned failures. The impact of server outages is minimized, and users can continue with their business simply by reconnecting to the server cluster.

LAWSON INSIGHT and the Informix Dynamic Server have been installed on an RS/6000 HA50 high-availability system. The HA50 offers a compact package of two H50 nodes with redundant Serial Storage Architecture (SSA) disk controllers and network adapters to provide maximum availability to Lawson and Informix users. The two nodes of the HA50 share a drawer of IBM SSA disks in a mirrored configuration.

The HA50 system running HACMP provides the following:

- Continuous Lawson and Informix availability without significant interruption to users if an SSA serial disk, disk adapter, or network adapter fails
- Fast response failover of approximately two minutes if there is a catastrophic hardware failure of one of the H50 nodes; provides minimal downtime for Lawson and Informix users while components are replaced or software problems are resolved

LAWSON INSIGHT

With the LAWSON INSIGHT Business Management System, Lawson Software provides three categories of solutions designed to manage global enterprise performance.

Business Process Solutions including Financials, Enterprise Budgeting, Human Resources, Supply Chain, and Collaborative Commerce streamline organizational process and foster operational excellence.

Value Management Solutions™ including Analytic Suite and Financial Performance Management enhance strategic decision making.

James T. West

Open Component Solutions™ allow users to leverage existing skills within the organization by selecting the interface development tools and presentation options that best meet organization—and individual—preferences. It is possible to view and tailor desktops, forms, charts, and graphs using LAWSON INSIGHT Open Component Solutions including Open Component Desktops and Open Component Tools for Java™, Microsoft® ActiveX®, Lotus® Domino™, and JavaScript™/HTML.

Lawson technology includes the following:

Self-Evident Applications™ (SEA). Navigating enterprise applications is easy and economical. SEA leverages user-friendly Web displays to eliminate costly and time-consuming training, documentation, and PC-based software maintenance. Self-service modules enable employees to directly participate in relevant company processes, eliminating costly and time-consuming data entry.

Drill Around™. Lawson's signature feature enables you to navigate from summary statistics to specific details, penetrate complex data relationships, and track information back to its original source (even an external database) at the click of a mouse.

Web-Deployable. Lawson Software was the first company to give its customers the ability to securely access business-critical enterprise information over the Web, from anywhere at anytime, running only a browser.

Open Systems. LAWSON INSIGHT solutions run on a wide variety of hardware platforms using the most advanced operating systems, so you can choose the computing platforms, systems, and presentation options that best meet your needs. You also may view your enterprise data with presentation options including Lotus Domino, Microsoft ActiveX, Java, and JavaScript/HTML.

Object-based Architecture. Lawson's open, multi-tier architecture provides a flexible, cost-effective mechanism for supporting Web-based computing; dramatically reduces time and expense of deploying new or updated software across the entire organization; and offers better speed and system performance. Lawson's object-based architecture also provides the foundation for Open Component Solutions, presentation options, and tools for customizing desktops, forms, charts, reports, and graphs.

Lawson's open, multi-tier architecture provides a flexible, cost-effective mechanism for supporting Web-based computing.

INFORMIX

Informix Dynamic Server is a powerful, multithreaded database server designed to deliver database scalability, manageability, and performance—while exploiting the capabilities of uniprocessor, symmetric multiprocessor (SMP), SMP clusters, and massively parallel processing (MPP) architectures. Informix Dynamic Server's core technology is based on the Dynamic Scalable Architecture™, which provides the most advanced parallel database architecture available for empowering today's distributed enterprise—from the desktop to departments to the data center. Dynamic Scalable Architecture is designed explicitly for managing increasingly larger and more complex databases while substantially improving overall system performance and scalability.

Key advantages of Informix Dynamic Server include:

- Maximum performance and scalability through a superior multithreaded parallel processing architecture
- Reduced operating system overhead through bypassing operating system limits its local table partitioning for superior parallel I/O (input/output) operations and high-availability database administration
- Parallel SQL functionality increases performance and lets all database operations execute in parallel, thereby eliminating potential bottlenecks

- High database availability for supporting a wide range of business-critical applications on open systems platforms
- Dynamic, distributed online system administration for monitoring tasks and distributing workloads
- Full-feature parity on Windows NT® and UNIX® operating systems
- Full relational database management system (RDBMS) functionality across all hardware architectures (uniprocessor, symmetric multiprocessing, and cluster systems) and database models (relational and object relational) enables seamless migration of applications, data, and skills

HACMP for AIX

HACMP for AIX offers a low-cost, efficient way to guard against unplanned outages and server downtime by providing a hot-standby capability. HACMP for AIX is useful for planned outages to allow for major configuration upgrades or routine hardware maintenance, while minimizing the impact to the server's users.

The HACMP for AIX design in the HA50 environment centers around a cluster manager that checks the status of cluster components in the HA network and negotiates with the nodes inside the HA50. The HA Cluster Manager allows the H50 nodes to collectively detect and act on changes in the environment defined to HACMP for AIX.

If a hardware or software failure occurs, HACMP's cluster manager starts a reconfiguration process, making the data and services of the failed node available to clients as transparently as possible. HACMP for AIX can be configured to automatically notify the system administrator of a variety of system events. This capability is important, since many system events requiring administrator attention will be transparent to users.

IBM HA50

The IBM HA50 is a rack-mounted system comprised of two RS/6000 H50 nodes having redundant power supplies, cooling fans, power cables, SCSI controllers, boot disks, network adapters, and SSA adapters and cables. The two H50 nodes in the HA50 share control of an IBM 7133-020 SSA disk array composed of eight SSA disks. The IBM HACMP software for AIX offers users several different configuration options for server failover and recovery. LAWSON INSIGHT could be installed on one HA node with Informix Dynamic Server installed on the other, or both Lawson and Informix products could be installed on the same HA node with the other node available as a hot standby.

The 7133 Serial Disk System is designed for high availability. With its redundant loop design, a single cable failure will not cause loss of access to data. If there is a failure on a loop, the server disk adapter is designed to continue accessing the disk.

LAWSON-INFORMIX-IBM: High-Availability Cluster

Many Lawson installations are running mission-critical applications where user response is required 24 hours a day, 7 days a week. In these installations, downtime can have a significant impact on people depending on system availability to meet mission-critical needs. IBM's high-availability technology can meet the needs of Lawson customers for mission-critical applications by providing protection from loss of data and redundant system features to minimize system downtime to provide a dependable and reliable critical system resource.

> ***HACMP for AIX can be configured to automatically notify the system administrator of a variety of system events.***

IBM has worked with Lawson and Informix in installing, testing, and certifying Lawson and Informix in IBM HA's ClusterProven technology. At the Lawson Software Conference and User Exchange, both LAWSON INSIGHT and Informix Dynamic Server run on the same node of the two-node HA50. If that active node fails, the other "hot-standby" node takes over the host name, IP address, and

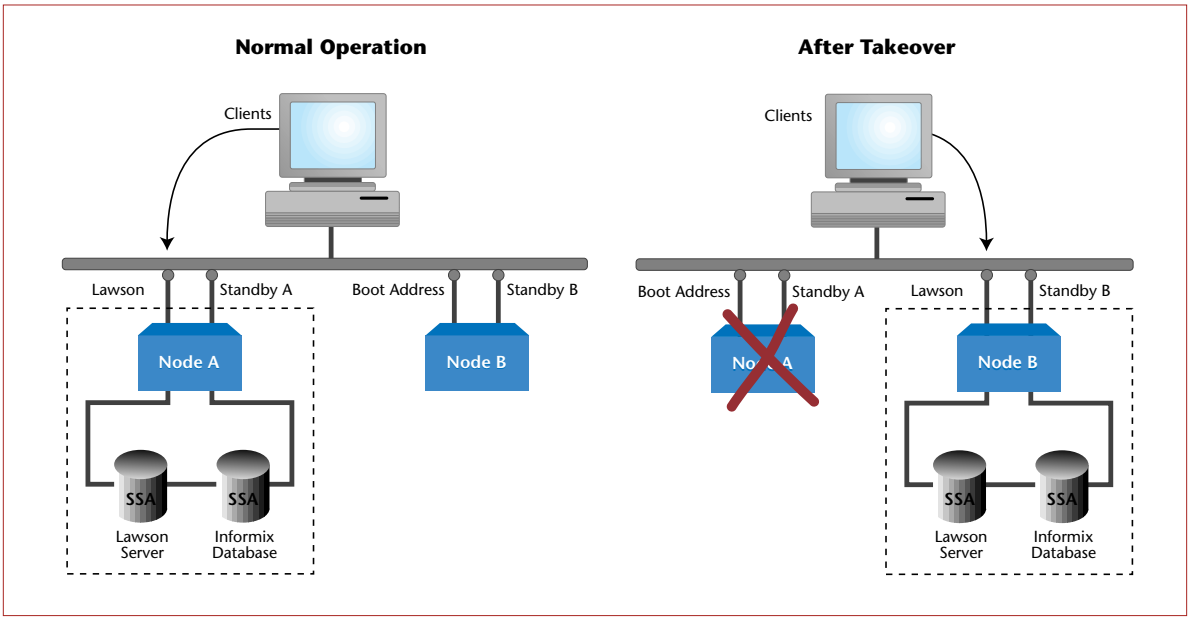


Figure 1. Lawson-Informix node takeover with HACMP

hardware MAC address of the cluster. This process occurs automatically without manual intervention. The user needs only to be aware of a single server hostname and IP address. During an HA failover, the user is only aware of a temporary system outage of a couple of minutes while the failover server takes over the services of the cluster. After the server has performed the failover, users may reconnect and continue their work.

The Lawson-Informix demo installed on the IBM HA50 runs a simplified payroll cycle, which loads an Informix database with simulated employee data and submits several payroll jobs to compute the company payroll. During the execution of the payroll jobs, a catastrophic failure is simulated causing LAWSON INSIGHT and Informix to abruptly stop executing in the middle of running the payroll jobs.

During this simulated system failure (see Figure 1), the standby HACMP node takes over the system and hardware resources of the failed machine, brings up the Informix database, starts the LAWSON INSIGHT servers, and is available for the system administrator to submit the jobs awaiting recovery.

The entire failover process takes place in less than two minutes. During this time, a system administrator is notified to take the appropriate course of action.

Conclusion

LAWSON INSIGHT using Informix Dynamic Server database software running on the IBM HA50 platform provides users with a proven high-availability system to meet mission-critical needs for 24x7 availability for the most demanding server needs where downtime is disastrous.

***James T. West**, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. West joined the IBM Federal Systems Division to work with the Superconducting Super Collider Laboratory, then continued with IBM Federal Systems performing scientific technical benchmarking on the SP. In 1995, he joined the RS/6000 Division to work with large commercial software vendors in software enablement and benchmarking. Mr. West has a BA in Physics from Emory University and an MS in Nuclear Engineering from Georgia Institute of Technology.*

PARTNER FOCUS

SAS and GPFS—A Scalable Solution

By Keith F. Olsen and James T. West

IBM performed several tests to determine functionality between SAS and GPFS and to measure performance and scalability of the GPFS file system. The results showed that GPFS is an excellent candidate for solving problems introduced by large numbers of SAS users or jobs in which data needs to be shared, or where large amounts of I/O activity can be helped through the addition of disks and adapters.

IBM has introduced a general-purpose parallel file system called the General Parallel File System (GPFS), which runs on the RS/6000® SP™. One concept actually drives parallel implementations for all systems: spread workload across many nodes in order to scale up the amount of work that can be done, while simultaneously delivering excellent performance.

Since many SAS users run their applications on the AIX® platform, we decided to determine whether the SAS system could benefit from using GPFS on the RS/6000 SP. Our project goals were twofold:

- To prove that GPFS and SAS were functional
- To measure performance characteristics and scalability

We ran the test project at the RS/6000 Teraplex Integration Center in Poughkeepsie, New York. The Teraplex Center, which focuses on large-scale business intelligence (BI) environments, provides testing facilities for customers in a real-world environment. The Teraplex Integration Center helps existing IBM customers, potential new customers, business partners, and IBM hardware and software developers by integrating, optimizing, and stressing BI systems on gigabyte- to terabyte-class databases.

This article summarizes the hardware and software configuration, the tests, test results, and performance on the RS/6000 SP. It includes an overview of GPFS that describes the capabilities of GPFS, a high-level description of GPFS, and how it was configured in these tests. It also includes our experiences in setting up and using GPFS in the SAS environment.

Keith F. Olsen

James T. West

GPFS Overview

IBM General Parallel File System for AIX (GPFS) provides file-system services to both parallel and serial applications running on the RS/6000 SP. GPFS allows users shared access to files that can span multiple disk drives on multiple SP nodes.

GPFS supports AIX file-system utilities. All commands that are currently used by AIX users can continue to be used—

unchanged for GPFS file-system users. The only unique commands, those for administering the GPFS file system, are available through the automated help panels.

GPFS allows parallel applications to simultaneously access the same—or different—files from any node in the configuration, while managing a high level of control over all file-system operations. It offers extremely high recoverability while maximizing data accessibility.

GPFS improves system performance in several ways:

- Allows multiple processes or applications on all nodes of the SP to simultaneously access the same file
- Increases aggregate bandwidth of the file system by spreading reads and writes across multiple disks
- Balances the workload evenly across all disks to maximize their combined throughput
- Supports large amounts of data and allows larger file systems by removing the physical limits of a single node
- Enables concurrent reads and writes to the file system

GPFS uses a sophisticated token management system to guarantee data consistency while providing multiple, independent paths to the same file, by the same name, from anywhere within the SP system.

GPFS is a logging file system that creates separate logs for each SP node. These logs record the allocation of metadata and aid fast recovery and data consistency if a node fails, even if a node fails while modifying file data.

GPFS allows disks to be added or deleted while the file system is mounted. When the time is right and system demand is low, you can rebalance the file system across all currently configured disks.

Hardware Configuration

We used an eight-node RS/6000 SP at the Teraplex Center for the SAS/GPFS tests. Each of the eight nodes was an identically configured symmetric multiprocessor (SMP) called a high node. The eight nodes spanned two SP frames, and all the nodes were connected by the SP Switch, a scalable, high-speed, high-availability network.

Each high node had the following configuration:

- Eight 604 112 MHz PowerPC™ processors
- 2 GB memory, two 2.2 GB internal disks
- Two Serial Storage Architecture (SSA) adapters
- Sixteen 4.5 GB SSA disks
- Four loops (two loops per adapter) with four disks on each loop

Software Configuration

For our tests, we used the following software configuration:

- AIX Version 4.2.1
- Parallel Systems Support Program (PSSP) 2.4
- GPFS Version 1.1
- SAS System Version 6.12

Physical Disk Configuration

Each node had an identical physical disk configuration of two SSA adapters, with two loops on each adapter and four disks on each loop. Figure 1 shows the configuration.

System Configuration

Each test was run against three types of file systems: local, NFS mounted across the SP

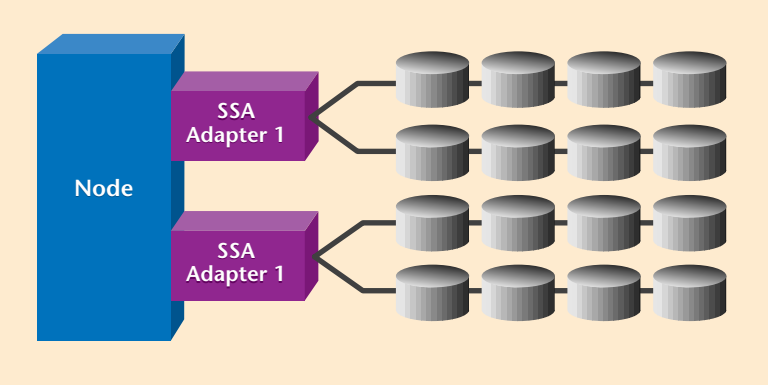


Figure 1. Node configuration

Switch, and GPFS. The first set of tests were run against an identical configuration of disks on local, NFS, and GPFS. The second set of tests were run on GPFS, where the number of disks and adapters were four times the size of the initial configuration. The disk configurations for each of the different file systems are described as follows:

Local Disk Configuration. Two file systems were defined in each file-system type:

- /DATAFS was used to store the newly created Household and Person data files
- /TEMPFS was used as the SASWORK directory

The disk configuration for the local file system was six 4.5 GB SSA disks for both /DATAFS and /TEMPFS—a total of 12 disks. Each node of the system configuration had two SSA adapters with 16 disks and two loops on each adapter with four disks on each loop. The file systems used 12 of the 16 disks. Figure 2 shows the local and NFS configurations.

NFS Disk Configuration. The local file systems were NFS mounted across the SP Switch for the NFS tests. The tests were run on one and four nodes; each node differed from the node on which the disk was locally attached. See Figure 2 for the NFS configuration.

GPFS Disk Configuration. The GPFS disk configuration was identical to the local configuration because each /GPFS/DATAFS and /GPFS/TEMPFS file system was six 4.5 GB disks on a single node containing two SSA adapters with two loops on each adapter. Figure 3 shows the single-node GPFS configuration.

The second GPFS disk configuration had four nodes, with each node containing six 4.5 GB disks for /GPFS/DATAFS and /GPFS/TEMPFS. This second configuration was four times the size of the initial GPFS configuration and four times the size of the local and NFS configurations. Figure 4 shows the four-node GPFS configuration.

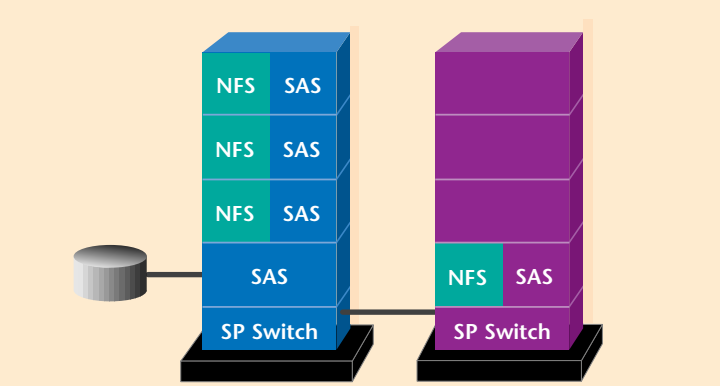


Figure 2. Local and NFS configurations

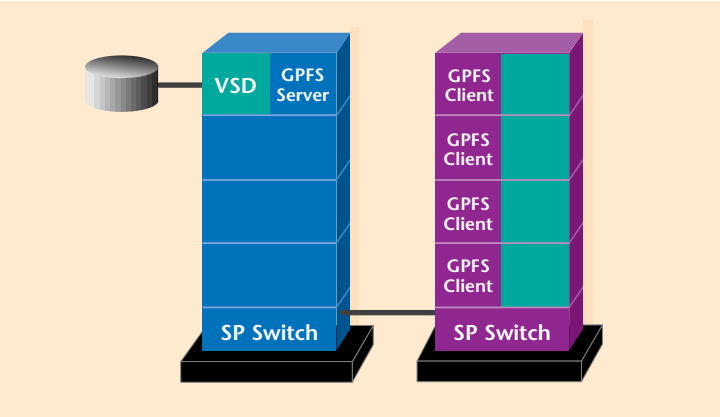


Figure 3. Single-node GPFS configuration

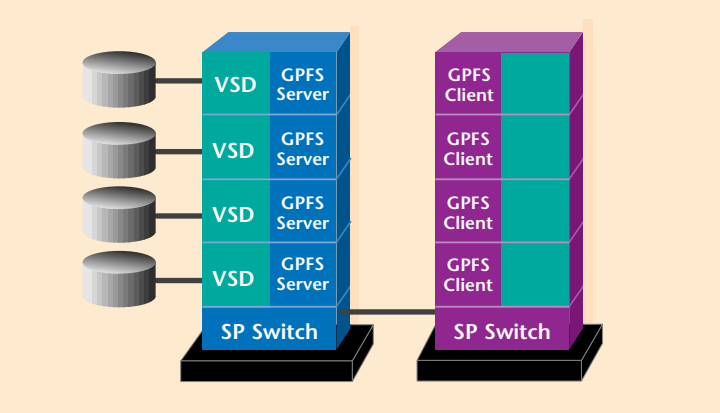


Figure 4. Four-node GPFS configuration

GPFS Test Cases

The four GPFS test cases included the following:

Census_Test1.sas. This case reads the raw data supplied from the U.S. Bureau of Census files and creates two SAS datasets:

HRECS and PRECS. The HRECS file generated using data from the State of California is 120 MB; the PRECS file from the State of California is 340 MB. The HRECS SAS dataset represents attributes describing households. The PRECS SAS dataset describes attributes for individual persons. The other three SAS test cases use the HRECS and PRECS SAS datasets created by the `Census_Test1.sas` input file.

Census_Test2.sas. The second test case creates indexes in the HRECS and PRECS SAS datasets that allow the two files to be joined in a later SAS test case. Although generating indexes is very memory intensive, it allows more efficient data access inside a SAS dataset.

Census_Test3.sas. This statistical test generates frequency data and summary data for HRECS and PRECS. Many PROC FREQ steps are executed on many combinations of data in the HRECS dataset. Many PROC SUMMARY steps are executed on data in the PRECS dataset. This test is very I/O intensive.

Census_Test4.sas. This test, which is very CPU intensive, collects frequency data on the PRECS dataset. It consists of two PROC UNIVARIATE steps: a PROC SORT step and two PROC data steps for cleanup.

GPFS Test Results

The test results are shown in graph formats for each of the four tests.

These results demonstrated two important points: GPFS functions well with the SAS System, and certain production environments can greatly benefit from using it. Figures 5, 6, 7, and 8 show the response times for the four census tests. Each graph has three sets of bars:

1. The first group indicates the time for one job running on Local, NFS, and GPFS file systems.

2. The second group indicates the average times for four concurrent jobs running: one on each of four separate nodes for NFS and GPFS file systems.

3. The third group shows the average times for 16 concurrent jobs running: four on each of four separate nodes for NFS and GPFS file systems.

The results from the third group clearly show the benefits of using a GPFS file system. The average response times of 16 concurrent GPFS jobs are significantly better than those for the NFS file system. These results are logical because we have scaled up the number of disks, adapters, and I/O processors to handle the I/O portion of the workload for the GPFS tests, while the I/O portion of the workload for the NFS tests must be handled by one node. As the workload becomes more I/O intensive, this benefit will be more pronounced because the GPFS file system will be able to provide the same performance while servicing a larger number of nodes, applications, and users.

Results from Test 1

Figure 5 shows the results from Test 1, including the three sets of results.

Each Test 1 job reads from the same input file and creates a new Household (HREC) and Person (PREC) file.

The first set of results, represented by the left three bars, indicates the time to run one job on local, NFS, and one-node GPFS file systems. The results show that while GPFS is slightly better than NFS, the local file system is the better solution for a single job.

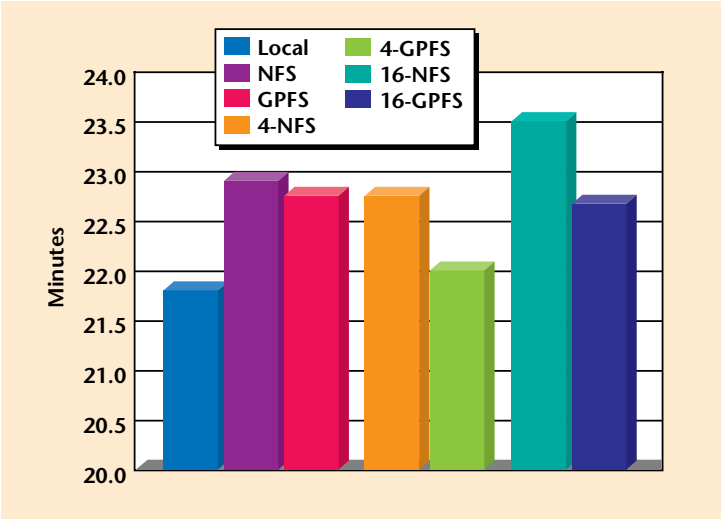


Figure 5. Test 1 results

The second set of results, indicated by the middle two bars, indicates the time to run four concurrent jobs on NFS and GPFS. The four concurrent jobs use four nodes, each node running only one job. In this case, GPFS generally performs better than NFS.

The third set of results, represented by the right two bars, indicates the time to run 16 concurrent jobs on NFS and the four-node GPFS file system. Four nodes are used, each node running four concurrent jobs. The results for this higher workload indicate again that GPFS performs much better than NFS.

Results from Test 2

Figure 6 shows the three sets of results from Test 2. Each Test 2 job sorts both the Household (HREC) and Person (PREC) files, then creates indexes to join the files together.

The first set of results, represented by the left three bars, indicates the time to run one job on local, NFS, and one-node GPFS file systems. We again see slightly better performance on the local configuration compared to NFS and GPFS.

The second set of results, indicated by the two middle bars, indicates the time to run four concurrent jobs on NFS and GPFS. The four concurrent jobs use four nodes, each node running only one job. This test shows no real difference in performance.

The third set of results, represented by the right two bars, indicates the time to run 16 concurrent jobs on NFS and the four-node GPFS file system. Four nodes are used, each node running four concurrent jobs.

As seen from the graph, the time to complete the task grows slightly as the workload increases. When the workload is increased to 16 concurrent jobs, the NFS configuration performance is greatly affected. The GPFS configuration sees little or no negative effect to the increased workload, maintaining the same relative response times to accomplish the tasks.

Results from Test 3

Figure 7 shows the three sets of results from Test 3. Each Test 3 job does sequence calculations and summaries.

The first set of results, represented by the left three bars, indicates the time to run one job on local, NFS, and one-node GPFS file systems. The trend discussed above continues where the local configuration performs best.

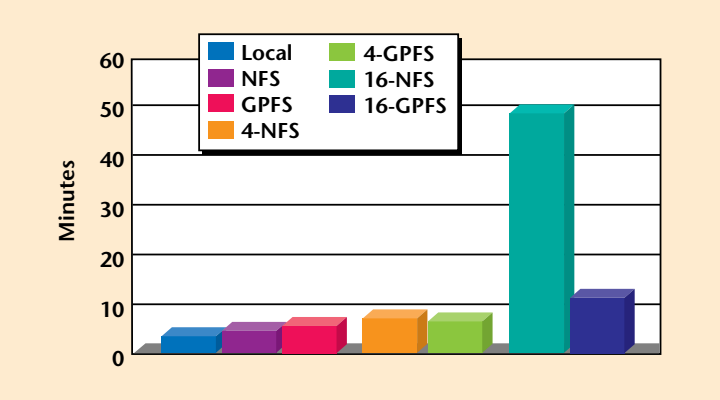


Figure 6. Test 2 results

The second set of results, indicated by the middle two bars, indicates the time to run four concurrent jobs on NFS and GPFS. The four concurrent jobs use four nodes, each node running only one job. In this test, NFS holds a slight advantage over GPFS.

The third set of results, represented by the right two bars, indicates the time to run 16 concurrent jobs on NFS and the four-node GPFS file system. Four nodes are used, with each node running four concurrent jobs. We see the GPFS advantage again in this test case. NFS performance is negatively affected, while GPFS has consistent results, despite the increased workload.

Results from Test 4

Figure 8 shows the three sets of results from Test 4. Each Test 4 job collects statistics on the files.

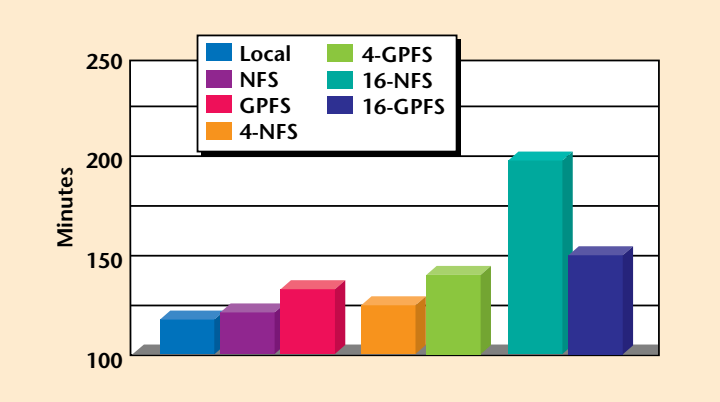


Figure 7. Test 3 results

The first set of results, represented by the left three bars, indicates the time to run one job on local, NFS, and one-node GPFS file systems.

The second set of results, indicated by the middle two bars, indicates the time to run four concurrent jobs on NFS and GPFS. The four concurrent jobs use four nodes, each node running only one job.

The third set of results, represented by the right two bars, indicates the time to run 16 concurrent jobs on NFS and the four-node GPFS file system. Four nodes are used, each node running four concurrent jobs. This set of results demonstrates the severe penalty paid by the NFS file system with a high workload. The results also show how scaling up the number of resources to handle I/O with GPFS can support the larger workload with ease.

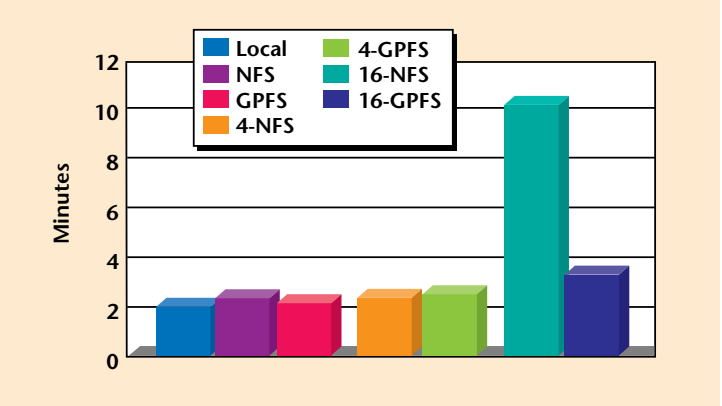


Figure 8. Text 4 results

GPFS Setup

This test project was our first opportunity to work with GPFS. We referred to three sources of information: a Redbook on GPFS, a white paper about performance with GPFS, and the GPFS Installation and Administration manual (see "For More Information" at the end of this article).

A few key parameters dramatically affect system performance. Since GPFS uses the SP Switch, certain parameters must be set in order to achieve maximum performance. In addition to switch tuning, certain system parameters, such as GPFS block size and Virtual System Disk (VSD) buddy buffers, must be set accordingly.

One key item to note is that several of these parameters came from the kernel memory (heap), which is limited in AIX 4.2.1; AIX 4.3 removes this limitation. We scaled down the number of buddy buffers to achieve a proper switch tuning setting. We used the parameter settings shown in Figure 9.

Although the GPFS block size was changed to 64 KB, it had little or no positive effect on performance.

After running the initial set of tests with a one-node GPFS configuration, we used the System Management Interface Tool (SMIT) to change the GPFS configuration. We supplied a file that contained the physical disk

Switch:

thewall=65536
rpoolsize=16777216
spoolsize=16777216

GPFS:

blocksize = 256K

VSD: Defined with the following parameters:

IP Packet Size	Initial Cache Buffers	Maximum Cache Buffers	VSD Request Count	rw Request Count	Buddy Buffer Minimum Size	Buddy Buffer Maximum Size	Buddy Buffer size:# maxbuf
61440	64	256	256	48	4096	262144	33

Figure 9. Parameter settings

descriptions to define the VSDs. The SMIT panel allows you to specify this disk descriptor file; it also uses this file to automatically build and configure the VSDs.

Once the VSDs were configured, the file system was automatically changed to include disks on all four nodes. Any data already in the file system was also automatically rebalanced across the entire file system using the 64 KB stripe size.

Although we did not use this option, GPFS allows the file system to be enlarged while still online.

Conclusions

We achieved our two test objectives. Our first goal, to test functionality between SAS and GPFS, was demonstrated because all of the tests ran successfully, with no changes required to any of the test cases. A comparison of the output files to the local, NFS, and GPFS tests were identical. No problems existed by having many SAS users on separate nodes accessing the same file in the GPFS file system.

Our second goal—to measure performance and scalability of the GPFS file system—was also accomplished. The results showed that scaling up the number of disks, loops, disk adapters, I/O buses, and processors allows more work to be done with comparable response times. The ability to access a file from any node in an SP system also allows CPUs to be scaled up and memory to handle the SAS workload.

GPFS can provide SAS users with the ability to define a large SASWORK directory. Defining a SASWORK directory in GPFS allows for an accessible file system to any node on an SP complex and simple, automated administration of that directory. System administration of a GPFS file system is exceptionally easy, especially when compared to administering a similar NFS configuration. The added benefits of a GPFS configuration are the availability and recoverability through the use of RAID, Logical Volume Manager mirroring, as well as data and metadata replicas inside GPFS.

The ability of GPFS to add disks to a file system and rebalance data across those new disks offers balanced, scalable growth to a file system. This ability to grow the file system in terms of processors, buses, disks, and adapters is unprecedented. It offers the ability to maintain acceptable performance while simultaneously increasing workload.

Our tests indicate that GPFS is an excellent candidate for solving problems introduced by large numbers of SAS users or jobs in which data needs to be shared, or large amounts of I/O activity can be helped by adding disks and adapters.

For More Information

Several sources for additional information about GPFS are listed below.

Papers

See *An Introduction to GPFS R1*, a white paper, on the Web at *www.austin.ibm.com/resource/technology*

Manuals

GPFS: A Parallel File System. An IBM Redbook. SG24-5165.
GPFS Installation and Administration. An IBM Manual. SA22-7278.

RS/6000 Teraplex Integration Center

For more information about the RS/6000 Teraplex Integration Center, contact Frank DeRobertis at *DEROB@us.ibm.com*

Keith F. Olsen, *IBM Corporation, 522 South Road, Poughkeepsie, NY 12601. Mr. Olsen is an AIX technical consultant providing technical support for Software Developer Partners for the RS/6000 SP, specializing in the Business Intelligence and Enterprise Resource Planning segments. He has a BA in Computer Science from State University of New York at Potsdam (SUNY Potsdam).*

James T. West, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. West joined the IBM Federal Systems Division to work with the Superconducting Super Collider Laboratory, then continued with IBM Federal Systems performing scientific technical benchmarking on the SP. In 1995, he joined the RS/6000 Division to work with large commercial software vendors in software enablement and benchmarking. Mr. West has a BA in Physics from Emory University and an MS in Nuclear Engineering from Georgia Institute of Technology.*

ClusterProven Program for RS/6000

Highly available systems provide the core of many large and small enterprise computing requirements. Whether a system is required to operate 24 hours per day 7 days per week (24x7) or 8 hours per day 5 days per week (8x5), unplanned downtime may result in significant losses and cannot be tolerated.

New Program for IBM Server Brands

The IBM Server Group has designed the ClusterProven program to partner with software developers to identify high-availability applications. The intent is to certify applications that have been designed to provide meaningful availability and scalability benefits for these demanding environments.

Each server brand—AS/400®, RS/6000®, S/390®, and Netfinity®—has a defined set of criteria that each application must meet to be ClusterProven. Solution developers who can demonstrate that their applications adhere to these design and implementation criteria may get their applications registered as ClusterProven. RS/6000 solution developers may seek certification at either of two levels: ClusterProven or Advanced ClusterProven.

Program Benefits

This program provides real benefits to both solution developers and end customers who run businesses with ClusterProven applications.

- **Developers who already provide products in this arena** will have the opportunity to be acknowledged. Currently, a large repertoire of applications are installed at customer sites in a High Availability Cluster MultiProcessing (HACMP) environment. These applications will most likely be the first certification candidates and will usually achieve ClusterProven status with minimal effort.

- **Developers whose customers have asked for a highly available product** will be guided toward that goal. Many vendors are unsure of what steps need to be taken to run their applications in a clustered environment. The ClusterProven program can relieve this cluster anxiety by providing a simple roadmap, beginning with basic concepts and arriving at ClusterProven status.

- **Customers** who have a ClusterProven product installed will have the assurance that the product has been tested in a clustered configuration and that it has successfully completed a test plan written by experienced HACMP specialists. Customers and service providers may find the documented results of the application certification useful in the implementation of their individual solution.

ClusterProven Certification Criteria

To become certified as ClusterProven on an RS/6000, an application must demonstrate successful exploitation of basic availability functions enabled through HACMP or HACMP/ES. Each application must be configured on an IBM server cluster and be validated against a standard testing methodology for the following types of events:

- Cluster node failure
- Partial data writes with data integrity preserved
- Adapter failure
- Network failure
- Device failure

After an IBM AIX/HACMP certified specialist reviews the resulting documentation, the application may be registered to be ClusterProven.

Advanced ClusterProven Certification Criteria

To be certified as Advanced ClusterProven on an RS/6000, an application must exploit an Advanced ClusterProven enabler that provides capabilities such as:

- Facilities to monitor application resources
- Application monitoring data in a usable format
- Monitoring facilities configurable for each application
- Scalability through workload balancing
- Detection of system and application workload
- Automated reallocation of workload
- Facilities to monitor, measure, and display availability

After the resulting documentation is reviewed by an IBM AIX/HACMP certified specialist, the application may be registered as Advanced ClusterProven.

Program Summary

The program has two levels of ClusterProven certification.

1. **ClusterProven.** This level leverages the features currently available in HACMP and HACMP/ES. It recognizes applications already exploiting this environment and aids in the development of other such applications.

2. **Advanced ClusterProven.** This level recognizes IBM and third-party enabler packages that proactively monitor, measure, and manage system resources. Applications that have already achieved ClusterProven status can be certified Advanced ClusterProven by providing monitoring and recovery information to an enabler program.

More Information

Program information can be found at *http://www.ibm.com/servers/clusters*.

AIX Questions

Compiled by Sandra Herzik

The Solution Developer Marketing Technical Support Group in Austin, Texas, supports software vendors who are developing or porting applications to AIX. This article is a compilation of questions that are frequently asked by vendors. The name of the responding Technical Support Group staff member appears after each response.

AIX Programming

How does VisualAge for Java Team Connection differ from other team development tools?

VisualAge® for Java™, Enterprise Edition, differs from other team development environments in the following ways:

1. Program elements are always available to everyone on the team since team developers do not reserve or "check out" these elements.

2. Incremental changes are immediately saved in the shared repository, so there is no need to "check in" a program element after it has been changed.

3. Anyone on the team can access and modify any program element for development, testing, and debugging purposes, regardless of who owns the program element. This facilitates code reuse and collaborative development.

4. Change is managed at the object level rather than the file level, which facilitates parallel development of classes by more than one developer.

5. Program element owners approve changes by releasing them into the team baseline. This emphasizes roles and responsibilities assumed by the team rather than file locking performed by the software.

VisualAge for Java's team features are optimized for object-oriented development in fast-moving, iterative, prototyping development environments. They are flexible and offer a high level of programmer productivity, while simultaneously providing stability.

—Anant Badrayani

Sandra Herzik

Anant Badrayani

Describe the environment variables used by VisualAge for Java.

The VisualAge for Java high-performance compiler uses several compile-time and runtime environment variables to set up its environment. The following environment variables are required at compile time:

CLASSPATH: Used by the hpj command at compile time to locate a class required during compilation. The runtime system class loader also uses this variable to locate a class compiled into a Dynamic Link Library (DLL) required by the compiled Java application.

TMP: Used by the hpj command at compile time to determine the directory under which it will place temporary files required by the compilation process. The default directory is /tmp.

IBMHPJ_HOME: Used by the high-performance compiler runtime system to locate the compiled versions of the Java class libraries that are supplied with the high-performance compiler. It is also used to locate property files and other files required by the run-time system.

IBMHPJ_OPTS: Used to pass runtime options to the high-performance compiler runtime system when an executable is being run.

LIBPATH: Used by the operating system to locate and load DLLs required by a running application.

—Anant Badrayani

What is the maximum repository size in VisualAge for Java?

Under AIX®, HP/UX, or Solaris®, the maximum repository size is 16 GB.

—Anant Badrayani

Why should I use the high-performance compiler?

Execution speed is important. To optimize speed, you may need a native code compiler for Java that will compile Java bytecode directly into native code in the same manner as compilers for C/C++, Fortran, COBOL, and other languages.

A Just-in-Time (JIT) compiler must balance time spent in optimizing code with time spent in executing that code. A native compiler, like the high-performance compiler, needs to compile code only once: before execution time. This enables it to apply resource-intensive optimization techniques to the generated code that are not available to a JIT compiler. These optimization techniques can greatly enhance the performance of the code. The degree of performance improvement, of course, is dependent upon your application.

The high-performance compiler allows you to write entire server applications in Java without sacrificing the benefits of platform-specific, native code performance. Cross-platform portability is retained through the Java bytecode or by compiling the bytecode to native code for each required platform.

—Anant Badrayani

How do I install the latest Fortran 6.1 for AIX?

Follow these steps to install the latest XL Fortran 6.1 Program Number 5765-D78:

1. Insert the CD into the drive.

2. Mount the CD using this command:

```
mount -o ro -v cdrfs /dev/cd0 /mnt
```

3. `cd /mnt/xlf/installp`

4. Type `smitty <hit enter>`, then follow this menu:
   - Software Installation and Maintenance
   - Install and Update Software
   - Install and Update from ALL Available Software
   - INPUT device / directory for software [.] (insert a period for current directory)
   - SOFTWARE to install []. Press F4 (Choose xlf products)
   - Press enter to install.

5. `cd / then umount /mnt`

—Wade Carlin

Wade Carlin

Can I use existing makefiles in VisualAge C++ for AIX?

VisualAge C++ (VAC) includes a tool called `make2cfg`, which converts the traditional `makefile` into a configuration file. Once this conversion is complete, you can build your program using the configuration file.

—Wade Carlin

Does AIX have a tool to check for Y2K compliance?

Yes, the tool can be downloaded from the following Web site:

http://www.software.ibm.com/year2000/papers/aixy2k.html#fixcheck

—Wade Carlin

How can I build and execute a VisualAge C program?

Follow the steps in Figure 1.

AIX System Administration

I must set the fast IPL option available on the service menu for my Micro Channel© multiprocessor system. However, this setting is lost after the system reboots, and I have to reset it. Is there a way to make this setting permanent?

The `mpcfg` command allows the root user to manage the multiprocessor's service information from the command line. To test a command-line setup of a fast IPL before you make it permanent, type the following command, then shut down the system:

```
mpcfg -cf 11 1
```

Adding the `-s` option to the `mpcfg` command places the changed values of information into */etc/lpp/diagnostics/data/bump* and stores them permanently. To restore the default service information from the */etc/lpp/diagnostics/data/bump* to NVRAM, you can use the `mpcfg -r` command.

—Sandra Herzik

VisualAge C program

(continued on following page)

Figure 1. VisualAge C program

VisualAge C program

(continued from previous page)

Program name: vacexample.c

Description: Shows users how to build and execute a VAC program on AIX 4.3.x using VAC 4.0

Function: Shows the use of the VAC build and runtime using a simple `hello.world` program. User interaction is needed to use this example code.

To build a VAC executable, you need two pieces. One is called an icc, similar to a make; the other is the c source file before you try to compile.

Example:

```
cd /tmp
vi a hello world ...



#include <stdio.h>

main ()
{
    printf ("We are ok if we see hello world\n");

    return;

}
```

Save the above as `hello.c`. Then create (`vi`) another file in the same directory called `hello.icc`.

```
option opt(level, "0")
{
   target "hello.exe"
   {
        source "hello.c"
   }
}
```

Save the above as `hello.icc`.

Comments for the above code

The directive defines options for one task involved in a build. For example, option directives select options that apply during the compile, link, and other steps of a build. This directive can be used to define local variables that will control the compilation and linking of the project. The value null or 0 indicates no specific options for the application.

When ready to compile, run from command line and enter the following:

```
vacbld hello.icc
```

(located in `/usr/vacpp/bin/vacbld`)

Once the executable is built, then run `./hello.exe`. A successful compile should print the "hello world" text.

Figure 1. VisualAge C program

What is the best approach for Y2K testing on AIX?

IBM recommends that customers use a non-production, stand-alone system to perform Year 2000 testing. Visit the following Web site for documentation: *http://www.software.ibm.com/year2000/papers/aixy2k.html*

Look in the section called Testing for Applications or Application "Chains." The documented procedures also state that "If you must use your production system for testing, be sure to use highly controlled environments after carefully studying and considering the impact of each testing scenario."

This document also highlights considerations for Y2K testing on SP™ systems. This is important because one fundamental design point for the SP system requires time-based synchronization of the SP nodes and the control workstation clocks. This SP example demonstrates the need for anyone doing Y2K testing to understand any time synchronization design requirements of the systems and the applications before attempting any Y2K-readiness test.

Be sure to note the potential hidden repercussions of moving time forward on your system and then moving time backwards. Moving time forward can normally be done if the requirements for both the system and application are met in moving time forward. This will ensure data integrity.

We recommend the following approach for Y2K testing:

1. Do a new install on the system with the machine clocks set to 12/../99. Remember to keep these systems separate from other non-test systems that might be misled by the difference in time/date. These test systems should not be connected to a production network.

2. Install the other applications on the system so they will use the 12/../99 date that is currently set.

3. Once the system installation and configuration is complete, create a system backup that can be easily reloaded to this state.

4. Let the system transition through the normal date change without manually changing the date or time.

5. Since 2000 is a leap year, be sure to include leap year testing.

6. For retesting, reload the systems from backup. If this system is to be returned to the production environment, reinstall the system using the current date and time.

—Brenda Hagler

Brenda Hagler

How does DCE affect Y2K testing of applications?

The Distributed Computing Environment (DCE) represents an example of a product designed with time-based synchronization requirements for each system participating in the distributed environment. The DCE design was based on time always going forward, so if DCE is one of the applications on your system or if you are running other applications that require DCE, you should follow recommendations for Y2K testing on these systems.

For systems that use DCE, you can move the date/time forward by stopping DCE, changing the date/time, then restarting DCE. Never move the date/time backwards. It is important to examine each application and system environment to understand the potential problems, recommendations, and required processes to move time forward. In addition, other considerations include the process and repercussions of moving time backwards for those systems and applications, and for ensuring data integrity.

Time can only move forward with DCE. Since time cannot "go backwards," we recommend using a non-production system for Y2K testing that involves any applications that require DCE. To retest a system, reinstall the system using the modified date and time. To place a system back into

production, reinstall the system using the current date and time.

DCE is Year 2000 ready. For more information, visit the DCE home page at *http://www.software.ibm.com/enetwork/dce/*

—Brenda Hagler

Compiled by Sandra Herzik. *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Ms. Herzik is a staff software engineer. She has a BA in International Relations and an MBA in Finance from American University in Washington, D.C.*

Solution Partnership Centers are a benefit of the IBM Solution Developer Program. The IBM Solution Developer Program provides marketing support to solution developers, helping them reach new customers and expand into new markets worldwide.

Explore the Power of the RS/6000 SP Server

If you are a commercial software developer and you want to know how your application runs on the new RS/6000 SP large scale server, visit the IBM Solution Partnership Center (SPC). At the SPC, you can explore the power and performance of the SP server for transaction processing and discover how the new version of AIX provides a solid foundation for a variety of application—including applications for the world of e-business and business intelligence.

The SPCs offer private labs to commercial software developers for porting, enabling, and testing their applications. These labs are equipped with a wide range of tools and technologies from industry-leading hardware and software providers. While visiting a Solution Partnership Center, developers can work with their applications on various platforms without investing in any additional equipment or staff. The SPCs also offer ongoing technical seminars and hands-on workshops related to critical technology issues.

For additional information about the Solution Partnership Centers, visit the Web site: http://www.spc.ibm.com/

Information about the IBM Solution Developer Program is available at the Web site: http://www.developer.ibm.com/

IBM

IBM Solution Partnership Center
2929 Campus Drive Suite 275
San Mateo, CA 94403
Phone: 650-524-5246
Fax: 650-524-5310
E-mail address: spcw@spc.ibm.com
Web: http://www.spc.ibm.com.

IBM Solution Partnership Center
404 Wyman Street – North Entrance
Waltham, MA 02154
Phone: 1-800-678-4249 or
781-895-2610

IBM Solution Partnership Center - Europe
Galileo Centre, Hursley Park
Winchester
Hampshire, S021 2JN
United Kingdom
Phone: 44-1962-816664