# TIBCO ActiveSpaces® Transactions

## Performance Tuning Guide

*Software Release 2.5.8*
*Published November 10, 2017*

Two-Second Advantage®

## Important Information

# Contents

# List of Figures

# About this book

This guide describes performance tuning of TIBCO ActiveSpaces® Transactions applications. This guide provides the information needed to design, monitor, and improve the performance of TIBCO ActiveSpaces® Transactions applications.

This book is intended for the following types of readers:

- TIBCO ActiveSpaces® Transactions application developers.
- System architects.

This guide is organized into these general areas:

- Description of the approach and technical background required to design and understand TIBCO ActiveSpaces® Transactions application performance. This information is in Chapter 2.
- Overview of TIBCO ActiveSpaces® Transactions features and how to use them to build a high-performance application. The trade-offs on using TIBCO ActiveSpaces® Transactions features are described. This information is in Chapter 3.
- Architectural and system configuration parameters that impact TIBCO ActiveSpaces® Transactions application performance. Scaling options are discussed. This information is in Chapter 4.
- Monitoring tools. System and application monitoring tools are discussed and the interpretation of the results and how to use it for performance tuning is described. This information is in Chapter 5.

This book is part of a set of TIBCO ActiveSpaces® Transactions documentation, which also includes:

**TIBCO ActiveSpaces® Transactions Installation —** This guide describes how to install the TIBCO ActiveSpaces® Transactions software.

**TIBCO ActiveSpaces® Transactions Quick Start —** This guide describes how to quickly get started using Java IDEs to develop TIBCO ActiveSpaces® Transactions applications.

**TIBCO ActiveSpaces® Transactions Architect's Guide —** This guide provides a technical overview of TIBCO ActiveSpaces® Transactions .

**TIBCO ActiveSpaces® Transactions Administration —** This guide describes how to install, configure, and monitor an TIBCO ActiveSpaces® Transactions deployment.

**TIBCO ActiveSpaces® Transactions Java Developer's Guide —** This guide describes how to program a TIBCO ActiveSpaces® Transactions application.

**TIBCO ActiveSpaces® Transactions System Sizing Guide —** This guide describes how to size the systems used to deploy TIBCO ActiveSpaces® Transactions applications

**TIBCO ActiveSpaces® Transactions Javadoc —** The reference documentation for all TIBCO ActiveSpaces® Transactions APIs.

# Conventions

The following conventions are used in this book:

**Bold —** Used to refer to particular items on a user interface such as the **Event Monitor** button.

**`Constant Width`** **—** Used for anything that you would type literally such as keywords, data types, parameter names, etc.

***`Constant Width Italic`*** **—** Used as a place holder for values that you should replace with an actual value.

# TIBCO ActiveSpaces® Transactions community

The TIBCO ActiveSpaces® Transactions online community is located at `https://devzone.tibco.com`. The online community provides direct access to other TIBCO ActiveSpaces® Transactions users and the TIBCO ActiveSpaces® Transactions development team. Please join us online for current discussions on TIBCO ActiveSpaces® Transactions development and the latest information on bug fixes and new releases.

# 1

# Introduction

## What is TIBCO ActiveSpaces® Transactions ?

TIBCO ActiveSpaces® Transactions is an in-memory transactional application server that provides scalable high-performance transaction processing with durable object management and replication. TIBCO ActiveSpaces® Transactions allows organizations to develop highly available, distributed, transactional applications using the standard Java POJO programming model.

TIBCO ActiveSpaces® Transactions provides these capabilities:

- Transactions - high performance, distributed "All-or-None" ACID work.
- In-Memory Durable Object Store - ultra low-latency transactional persistence.
- Transactional High Availability - transparent memory-to-memory replication with instant fail-over and fail-back.
- Distributed Computing - location transparent objects and method invocation allowing transparent horizontal scaling.
- Integrated Hotspot JVM - tightly integrated Java execution environment allowing transparent low latency feature execution.

## Approach

Performance must be designed into an application. It is not realistic to expect performance to be tested in at the end of the development cycle. A common sense approach is provided to design TIBCO ActiveSpaces® Transactions applications that perform. The key metrics and trade-offs that impact application performance are:

- Scaling versus path length
- Horizontal versus vertical scaling
- Contention
- Latency versus through-put

# Features

TIBCO ActiveSpaces® Transactions features make high-performance applications possible. However, these features must be used where appropriate. Each of these TIBCO ActiveSpaces® Transactions features can have an impact on application performance. They must be used when appropriate.

- Managed versus non-Managed objects.
- Transactional versus non-transactional code.
- Keys and indexes.
- Transactions versus Java monitors for concurrency control.
- High-availability (partitioned) Managed objects versus non-partitioned Managed objects.
- High-availability versus distribution.

# Tuning

Tuning a TIBCO ActiveSpaces® Transactions application involves both application architecture decisions and appropriate configuration of system parameters. The types of questions that must be answered are:

- How should the JVM heap sizes be configured?
- How to ensure that the garbage collector does not impact the latency of running applications?
- What size of shared memory should be configured?
- Should System V or file mapped shared memory be used?
- How much disk space is needed?
- Should the application execute on single or multiple nodes?
- How should the application be deployed within a node?
- Should application data be partitioned across multiple nodes?
- When should distribution be used?

# Monitoring

TIBCO ActiveSpaces® Transactions provides a rich set of application and system monitoring tools. These tools provide a way to monitor TIBCO ActiveSpaces® Transactions applications running

under a load to determine how to improve the performance. Standard JVM monitoring tools complement the TIBCO ActiveSpaces® Transactions tools.

# 2

# Approach

This chapter describes the design approach for TIBCO ActiveSpaces® Transactions applications. It also defines the key concepts that must be understood to perform performance analysis and tuning.

## Concepts

- Path length

  The amount of time that it takes to process a unit of application work (e.g. processing a request and sending its response), excluding any time spent blocked (e.g. disk I/O, or waiting for a response from an intermediate system).

- Horizontal scaling

  Adding more computing nodes (i.e. machines) to a system.

- Vertical scaling

  Adding more resources (e.g. CPUs or memory) to a single computing node in a system.

- Contention

  Competition for computing resources. When resources are not available the application waits and often uses up other system resources competing for the requested resource.

- Latency

  The time between when a request is issued and a response is received. Latency can consist of a variety of components (network, disk, application, etc...).

- Through-put

  A measure of the overall amount of work that a system is capable of over a given period of time.

# Guidelines

## Identifying performance requirements

- Clear and complete requirements.

  Start with a clearly stated set of requirements. Without this, performance work cannot be judged as either necessary or complete.

- What are the units of work to be measured?

  Request messages? Request and response messages? Some larger business aggregation of requests and responses? Are there logging requirements?

- Which protocol stacks will be used?

  How many connections? What are the expected request rates per connection? What do the messages look like?

- What are the Request/Respone latency requirements?

  What is the maximum allowable latency? What is the required average latency? What percentage of Request/Response pairs must meet the average latency? Occasionally there are no latency requirements, only throughput requirements.

- What is the sustained throughput requirement?

  How many units of work per second must be completed, while still meeting the average latency requirements?

- What is the burst throughput requirement?

  How many units of work per second must be completed, while still meeting the maximum latency requirements?

- Are third party simulators required for the testing?

  What role do the simulators play in the performance testing? What are their performance characteristics? Are they stable, correct, predictable, scalable and linear? Are they capable of generating a load that meets the performance requirements for the application?

## Measuring performance

Working on performance without first collecting meaningful and repeatable performance data is wasted effort. No one can predict the exact performance of an application, nor can one predict where the top bottlenecks will be. These things must be measured. And they must be re-measured as the application or environment changes.

- Develop an automated test.

  Performance testing involves the repeated configuration and coordination of many individual parts. Doing these steps manually guarantees a lack of repeatability.

- Measure meaningful configurations.

Do not test performance in the VMware® image. Test in an environment that is the same or similar to the production environment.

Use production mode binaries.

Test with assertions disabled.

Eliminate deadlocks from the application. The performance of a path which contains a deadlock is unboundedly worse than the same path without a deadlock. Performance tests which encounter deadlocks in the steady state are invalid, and indicate application problems.

Do not run performance tests with any extraneous code, logging or tracing enabled. Developers often add extra code and tracing to an application that may be of use in the development process. This code will not be used in the production system, and can significantly perturb performance measurements.

Use representative numbers of client connections. There is much to be learned from the performance characteristics of a single client talking to the application via a single connection. Performance testing should start with this case. However, TIBCO ActiveSpaces® Transactions is optimized for parallelism. Almost all well designed applications will support multiple connections. Performance testing configuration should mirror the actual application configuration. If the target configuration calls for 100 connections at 10 messages per second, per connection, test it that way. This is not the same as one connection at 1000 messages per second.

- Measure the steady state of an application, not the load stabilization time nor the application startup time.

  Complex systems generally have complex initialization sequences, and while there are often performance requirements for this startup time, they are generally not the primary performance requirements. Repeatable performance runs are done against an already started and stable application, with a warm-up period that allows for the test load to stabilize.

- Run on otherwise idle hardware.

  Steady states cannot be meaningfully measured if there is concurrent uncontrolled machine usage on the target system(s).

- Start measuring performance early in a project.

  Do not wait until the end of a project to create the performance tests. Performance measurement can, and should, be done throughout the life cycle of the project. Once a test is in place, it should be mostly a matter of configuration to integrate it with the application. Begin working with the real application as soon as possible.

- Performance runs versus data collection runs.

  Make a distinction between test runs which are measuring best performance, and those that are collecting data for analyzing performance. Best performance runs should have little or no extra data collection enabled.

- Don't measure saturated systems.

  When a system has reached the maximum rate of work that it can process, it is said to be saturated. Production systems are intentionally not run in this way, nor should performance testing be run in this manner. At the saturation point (or approaching it) systems can exhibit various forms of undesirable behavior; excessive CPU utilization per request, increased memory utilization, non-

linearly deteriorating response times and throughput. As the system nears the saturation point, the performance will generally decrease due to these effects. Saturation can occur at different levels, including protocol stacks, application logic, and the system itself. Performance testing should be organized to identify the various saturation points of the system.

For example, CPU utilization should not be driven to 100%. Typically tests should designed to drive the CPU utilization to a maximum of between 80 and 90%.

- Sweeping the load.

  Nothing useful can be gained by running a performance test with a single configuration that saturates the application. Proper performance testing calls for starting with a simple configuration and a modest workload that doesn't tax the application, and then increasing the load in subsequent runs to identify the linearity of the application and the saturation point. The exercise is then repeated with other configurations.

# Analyzing performance

In TIBCO ActiveSpaces® Transactions applications, we concern ourselves with three main types of performance:

- Single-path performance: the CPU cost and wall clock time for a single request.
- Multi-threaded or scaling: running the single path concurrently on multiple threads.
- Multi-node or horizontal scaling: running the single path concurrently on multiple threads on multiple machines.

We generally want to look first at multi-threaded performance. The TIBCO ActiveSpaces® Transactions runtime environment is optimized for running on multi-processor, multi-threaded platforms. Regardless of the single path speed, additional performance is most easily obtained by running the single path concurrently on multiple threads.

At this point you should have a set of data that describes the application functioning normally and in saturation. You will already have done some analysis that lead to your choice of configurations to measure.

Now look at your data asking scalability questions: Pick unsaturated data points with the same number of requests per second, and differing numbers of clients. How does the CPU utilization change as the number of clients are increased? If your data shows near-perfect linearity and scaling your application may not need tuning. In this case, additional performance can be gained by adding more or faster CPUs. Usually the data shows a lack of scaling or linearity, an inability to utilize all of the CPUs, or overall performance is not acceptable on the target hardware. The next task is to understand why. At this point, performance work resembles scientific research:

1. A set of experiments are run and data is collected.
2. The data is analyzed.
3. Hypotheses are made to explain the data.
4. A change is made to the system under test and the process is repeated.

At this point we transition to statistics collection runs to help us identify scaling bottlenecks. Typically scaling bottlenecks are a result of contention. Contention can be for a variety of resources; processing

cycles, network I/O, disk I/O, transaction locks, Java monitors, system memory, etc. Excessive garbage collection can also be a cause of performance problems. When trying to identify the cause of scaling problems there are no absolute rules which will tell us ahead of time which statistics reports will produce the most interesting data.

In looking at these data, one should first look for anything that indicates gross problems with the run, such as application failure, deadlocks or swapping. If seen, the results should be disregarded, and the problem corrected before continuing.

At this point, you should have an automated, repeatable test and data which demonstrate performance and/or scaling issues with your target application. You can now begin to use the collected data to optimize your application.

After you have removed the bottlenecks and the application scales well across multiple processors it may still not meet the performance requirements. The single execution path performance should be examined at this time with a Java profiling tool.

Horizontal scaling may also be examined at this point as a way to increase overall system throughput. Add a single node at a time (or pairs for High-availability active and replica nodes) to the test configuration and re-run the measurement process.

# 3

# Using product features

This chapter describes the key TIBCO ActiveSpaces® Transactions features and how to use them to ensure optimal application performance.

## Managed objects

**Features:**

- Transactional.
- Persisted in shared memory.
- Shared across multiple JVMs.

**Cost:**

Compared to a POJO, a Managed object will consume additional processing associated with providing transactional coherency, additional temporary shared memory resources associated with providing rollback capability, and shared memory associated with persisting the object.

**Usage:**

- As a replacement for placing object state in a database.
- To transactionally synchronize multi-threaded access to shared application data.
- To provide in-memory objects which can be navigated to with keys.
- When application state needs to be persisted across multiple invocations of the JVM.
- When application state needs to be shared between multiple JVMs.

**Avoid:**

- For temporary objects.
- For data which does not need to be persisted.

# Transactions

**Features:**

- Provide multi-reader, single writer object locking.
- May lock both Managed objects and transactional POJOs.
- Automatic deadlock detection.
- Automatic rollback of modifications to transactional resources when a deadlock or error is encountered.

**Cost:**

Additional processing for each object that is locked. Additional processing and temporary heap space for each transactional field which is modified. Additional processing for deadlock detection. Temporarily blocked threads when there is transaction contention.

**Usage:**

- Used to access Managed objects.
- May be used to transactionally isolate and protect modifications to POJOs.
- Used when multiple reader/single writer access to a resource is desired. Provides scalability for multiple readers executing simultaneously in multiple threads, while still providing data consistency through exclusive write locking.
- Small transactions scale better than large transactions.

**Avoid:**

- Using transactions to manage non-transactional resources (e.g. Network).
- Using transactions when transactional semantics for a resource are not required (e.g a counter that needs to atomically increment but never rollback).
- Deadlocks. Although the TIBCO ActiveSpaces® Transactions runtime automatically rolls back and replays deadlocked transactions, this is very expensive compared to avoiding the deadlock entirely. If deadlocks are seen in your testing, the involved code should be re-organized or re-written to eliminate the possibility of deadlock.
- Promotion locks. When two threads concurrently execute the same code path containing a promotion lock, a deadlock will be generated. Several different techniques can be used to eliminate promotion locks:

  Changing the code to take a write lock instead of a read lock at the first access in the transaction to the Managed object to be modified.

  When finding an object through a query, use either `LockMode.WRITELOCK` or `LockMode.NO-LOCK`.

When iterating objects from `ManagedObject.extent()` or `KeyQuery.getResults()` use either `LockMode.WRITELOCK` or `LockMode.NOLOCK`.

When the modification of an already read-locked object does not need to be done in the same transaction, move it to an `@Asynchronous` method and it will run in another transaction after the current transaction commits.

- Transaction lock contention. When a transaction is blocked waiting to acquire a lock, it remains blocked at least until the transaction holding the lock commits or aborts. It may remain blocked longer if there are multiple threads competing for the same transaction locks.
- Long running transactions. Transactional resources in multi-threaded applications are generally shared between threads. Locking a resource in a long running transaction can block other threads for the duration of the transaction. Short running transactions scale better than long running transactions.
- Large transactions (those that contain many locked resources). Large transactions tend to be more prone to generating contention and deadlocks. When there is contention between large transactions, even if there are no deadlocks, the deadlock detection becomes more expensive.

**Summary:**

Transactions are a powerful tool for maintaining application data consistency and scaling. But this feature comes at a cost. Avoid using transactions where they are not necessary.

# Java monitors

**Features:**

- Monitors (the Java synchronize keyword) provide a simple mutual exclusion mechanism.
- Lighter weight than transactions.
- Easy to cause undetected deadlocks.
- Multiple threads sharing read access to a resource become single-threaded when accessing the resource.

**Usage:**

- Use monitor when synchronization is required for non-transactional resources.

**Avoid:**

- Using monitors on transactional resources (they are already protected by transaction locking).

# READ_COMMITTED_SNAPSHOT Transaction Isolation Level

Use of this isolation level carries a performance penalty. An extra shared memory copy of the object data must be made the first time the data is accessed with a transaction. Subsequent accesses then use the read image, and commit frees the memory.

The default isolation level, SERIALIZABLE, does not carry this penalty.

# Keys and indexes

**Features:**

- Keys are only allowed on Managed objects.
- Allows the application to quickly and efficiently navigate to a unique Managed object or group of Managed objects.
- Supports unique, non-unique, ordered and unordered keys and queries.

**Cost:**

Each key requires additional processing resources at object creation time, additional shared memory resources.

**Usage:**

- Use keys as you would use an index in a database.
- Use unique keys instead of extent iteration for finding a single object.
- Use non-unique keys instead of extent iteration for finding a group of ordered or unordered objects.

**Avoid:**

- Using keys on objects that don't require navigation to find them.
- Defining unnecessary key fields.

# High-availability

**Features:**

- Transparent, transactional, high performance replication of object data across nodes.
- Transparent routing of data to a partition or node.
- High performance, automated support for migration of object ownership from a failed active node to a replica node.

**Cost:**

Additional CPU cycles and memory resources for managing the internal transaction resources when modifying a Managed object. Additional network I/O for propagating the modifications to the replica nodes.

Reads of highly available objects have the same performance as reads of Managed objects. No extra cycles are consumed and no network I/O is generated.

**Usage:**

- Use highly available objects to provide non-stop access to application data in the case of node failure.

- Use partitions with multiple replica nodes to provide a transparent, transactional push mechanism of object data to a group of nodes.
- Use highly available objects methods to execute behavior on the currently active node for a partition.
- Use highly available objects to transparently scale an application load horizontally across multiple nodes.

**Avoid:**

- Modifying highly available objects unnecessarily. Modifications cause network I/O and processing cycles on the replica nodes. If there is data being modified that is not necessary for the application to see after a fail-over, do not keep this data in a highly available object. Use either Managed objects or POJOs.

  Note, that in comparision to Managed objects and POJOs, a highly available object incurs extra processing costs even when there are no replica nodes defined for its partition.
- Making highly available objects larger than necessary. Each time a modification occurs, the entire object is copied to the replica nodes.
- Replicating object data to more nodes than is required. Each additional replica node requires additional network I/O and processing.
- For simple load balancing consider using a hardware based solution instead of the location transparent routing provided by highly available objects.

# Distribution

**Features:**

- Direct application access to the creation of remote objects and their data.
- Direct application access to remote method invocation.
- Optionally cached on remote nodes.

**Avoid:**

- Distributed deadlocks. Distributed deadlock detection uses a timeout to detect a deadlock. This implies that a distributed transaction will wait the entire value of the timeout value before a deadlock is reported. During this period of time the transaction is stalled.
- For simple load balancing consider using a hardware based solution instead of the location transparent routing provided by distributed objects.

# 4

# Tuning

This chapter describes how to tune TIBCO ActiveSpaces® Transactions applications. Application and system parameters are described.

## Deployment

The TIBCO ActiveSpaces® Transactions runtime supports multiple processes communicating through shared memory, or a memory mapped file. When a JVM is started using the deployment tool, all runtime resources required by the JVM are available in the same process space. There are cases where multiple JVMs on a single node may be appropriate for an application (see the section called “Multiple JVMs” on page 19), but there is a performance impact for dispatching between JVMs.

## JVM

### Heap size

By default, TIBCO ActiveSpaces® Transactions doesn't modify the JVM heap (`-Xms<size>` and `-Xmx<size>`) or stack (`-Xss<size>`) memory options. If during testing, the JVM is found to run short of, or out of memory, these options can be modified either setting them as arguments to the deployment tool.

Both `JConsole` and `VisualVM` can be used for looking at heap memory utilization.

### Garbage collection

By default, TIBCO ActiveSpaces® Transactions doesn't modify any of the JVM garbage collection parameters.

For production systems deploying using the Oracle JVM, we recommend that you enable garbage collection logging using the following deployment options:

- `-XX:+PrintGCDateStamps`
- `-XX:+PrintGCDetails`
- `-Xloggc:gc.log`

  Note: replace `gc.log` with a name unique to your deployed JVM to avoid multiple JVMs from colliding using the same log file.

This will be provide a relatively low overhead logging that can be used to look for memory issues and using the timestamps may be correlated to other application logging (e.g. request/response latency).

Another useful set of Oracle JVM option controls GC log file rotation. See (Java HotSpot VM Options [http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html]).

- `-XX:-UseGCLogFileRotation`
- `-XX:-NumberOfGCLogFiles`
- `-XX:GCLogFileSize`

Garbage collection tuning is a complex subject with dependencies upon the application, the target load, and the desired balance of application throughput, latency, and footprint. Because there is no best one-size-fits-all answer, most JVMs offer a variety of options for modifying the behavior of the garbage collector. An Internet search will show a large selection of writings on the subject. One book with good coverage on the implementation and tuning of garbage collection in Oracle JVMs is `Java Performance by Charlie Hunt and Binu John`.

As mentioned above, it is hard to make garbage collection tuning suggestions for a generic application environment such as TIBCO ActiveSpaces® Transactions . However, here is a comment from our field engineering organization with respect to tuning rules engine applications builtin on top of the \TIBCO ActiveSpaces® Transactions runtime:

We've found that a good garbage collection algorithm for a BE-X application is concurrent mark sweep ... although this is application dependent, it's probably a good start:

- `-XX:+UseCompressedOops`
- `-XX:+UseConcMarkSweepGC`

Another good collector, available in JDK 7 and later, is G1. Liveview testing showed it to perform well throughput optimization on systems that don't a have huge amount (more than 10 or 20 gigabytes) of heap to manage.

- `-XX:+UseG1GC`

# Out of Memory Heap Dump

When deploying using the Oracle JVM we recommend setting the following JVM deploy option which will cause a JVM heap dump to be logged upon an out of memory error within the JVM:

`-XX:+HeapDumpOnOutOfMemoryError`

## Multiple JVMs

Typically, an TIBCO ActiveSpaces® Transactions deployment will consist of a single JVM per node. However, there may be cases where multiple JVMs per node are required (e.g. Exceeding a per-process limit on the number of file descriptors).

TIBCO ActiveSpaces® Transactions supports multiple JVMs deployed within a single node. These JVMs may all access the same Managed objects.

Multiple JVMs are deployed by using the `detach` option to the deployment tool.

# Shared memory

- **Size**

  Shared memory needs to be large enough to contain all of the application's Managed objects, the runtime state, and any in-flight transactions. See the **TIBCO ActiveSpaces® Transactions Sizing Guide** for information on how to determine the correct size.

  When caching Managed objects, shared memory only needs to be large enough to store the subset of cached Managed objects.

- **mmap**

  By default TIBCO ActiveSpaces® Transactions uses a normal file in the file system. The `mmap(2)` system call is used to map it into the address space of the TIBCO ActiveSpaces® Transactions processes.

  In a development environment, this is very convenient. Many developers may share a machine, and the operating system will only allocate memory as it is actually utilized in the shared memory files. Cleanup of stranded deployments (where the processes are gone but the shared memory file remains) may be as simple as removing file system directories.

  A performance disadvantage when using mmap'd files for shared memory is that the operating system will spend cycles writing the memory image of the file to disk. As the size of the shared memory file and the amount of shared memory accessed by the application increases, the operating system will spend more and time writing the contents to disk.

  TIBCO ActiveSpaces® Transactions is not supported running from a networked file system (e.g. NFS).

- **System V Shared memory**

  TIBCO ActiveSpaces® Transactions also supports using System V Shared Memory for its shared memory.

  To reclaim System V Shared Memory the TIBCO ActiveSpaces® Transactions node must be stopped and removed using the `administrator remove node` command. The shared memory is not released by removing the node deployment directory.

  An advantage of using System V Shared Memory is that the operating system does not spend any cycles attempting to write the memory to disk.

Another advantage is that the memory is allocated all at once by the operating system and cannot be swapped. In some cases this also allows the operating system to allocate the physical memory contiguously and use the CPU's TLB (translation lookaside buffer) more efficiently. On Solaris this occurs automatically. See the section called “Linux Huge Page TLB support” on page 21 for Linux tuning information.

See the section called “Linux System V Shared Memory Kernel Tuning” on page 20 for details on tuning Linux System V Shared Memory kernel parameters.

## Caching

Managed objectss upport caching of a subset of the object data in shared memory. The cache size should be set so that it is large enough to allow a working set of objects in shared memory. This will avoid having to constantly refresh object data from a remote node or an external data store, which will negatively impact performance. TIBCO ActiveSpaces® Transactions uses a LRU (least recently used) algorithm to evict objects from shared memory, so objects that are accessed most often will remain cached in shared memory.

# Swapping

The machine where a TIBCO ActiveSpaces® Transactions node runs should always have enough available physical memory so that no swapping occurs on the system. TIBCO ActiveSpaces® Transactions gains much of its performance by caching as much as possible in memory. If this memory becomes swapped, or simple paged out, the cost to access it increases by many orders of magnitude.

On Linux one can see if swapping has occurred using the following command:

```
$ /usr/bin/free
             total       used       free     shared    buffers     cached
Mem:       3354568    3102912     251656          0     140068    1343832
-/+ buffers/cache:    1619012    1735556
Swap:      6385796          0    6385796
```

On Solaris, the following command can be used:

```
$ /etc/swap -l
swapfile             dev  swaplo blocks   free
/dev/dsk/c1t0d0s1   118,9      16 16780208 16780208
```

# Hardware Tuning

The BIOS for many hardware platforms include power savings and performance settings. Significant performance differences may be seen based upon the settings. For best TIBCO ActiveSpaces® Transactions performance, we recommend setting them to their maximum performance and lowest latency values.

# Linux Kernel Tuning

## Linux System V Shared Memory Kernel Tuning

Operating system kernels typically enforce configurable limits on System V Shared Memory usage. On Linux, these limits can be seen by running the following command:

```
$ ipcs -lm
------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 67108864
max total shared memory (kbytes) = 67108864
min seg size (bytes) = 1
```

The tunable values that affect shared memory are:

- SHMMAX - This parameter defines the maximum size, in bytes, of a single shared memory segment. It should be set to at least the largest desired memory size for nodes using System V Shared Memory.
- SHMALL - This parameter sets the total amount of shared memory pages that can be used system wide. It should be set to at least `SHMMAX/page size`. To see the page size for a particular system run the following command:

  ```
  $ getconf PAGE_SIZE
  4096
  ```

- SHMMNI - This parameter sets the system wide maximum number of shared memory segments. It should be set to at least the number of nodes that are to be run on the system using System V Shared Memory.

These values may be changed either at runtime (in several different ways) or system boot time.

Change SHMMAX to 17 gigabytes, at runtime, as root, by setting the value directly in /proc:

```
# echo 17179869184 > /proc/sys/kernel/shmmax
```

Change SHMALL to 4 million pages, at runtime, as root, via the sysctl program:

```
# sysctl -w kernel.shmall=4194304
```

Change SHMMNI to 4096 automatically at boot time:

```
# echo "kernel.shmmni=4096" >> /etc/sysctl.conf
```

# Linux Huge Page TLB support

On Linux, the runtime attempts to use the huge page TLB support the when allocating System V Shared Memory for sizes that are even multiples of 256 megabytes. If the support is not present, or not sufficiently configured, the runtime will automatically fallback to normal System V Shared Memory allocation.

- The kernel must have the hugepagetlb support enabled. This is present in 2.6 kernels and later. See ( http://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt [http://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt]).
- The system must have huge pages available. They can be reserved:

At boot time via /etc/sysctl.conf:

```
vm.nr_hugepages = 512
```

Or at runtime:

```
echo 512 > /proc/sys/vm/nr_hugepages
```

Or the kernel can attempt allocate the from the normal memory pools as needed:

At boot time via /etc/sysctl.conf:

```
vm.nr_overcommit_hugepages = 512
```

Or at runtime:

```
echo 512 > /proc/sys/vm/nr_overcommit_hugepages
```

- Non-root users require group permission. This can be granted:

  At boot time via /etc/sysctl.conf:

  ```
  vm.hugetlb_shm_group = 1000
  ```

  Or at runtime by:

  ```
  echo 1000 > /proc/sys/vm/hugetlb_shm_group
  ```

  where 1000 is the desired group id.

- On earlier kernels in the 2.6 series, the user ulimit on maximum locked memory (memlock) must also be raised to a level equal to or greater than the System V Shared Memory size. On RedHat systems, this will involve changing /etc/security/limits.conf, and the enabling the PAM support for limits on whatever login mechanism is being used. See the operating system vendor documentation for details.

# Linux ulimit number of processes tuning

A system imposed user limit on the maximum number of processes may impact to ability to deploy multiple JVMs concurrently to the same machine, or even a single JVM if it uses a large number of threads. The limit for the current user may be seen by running:

```
$ ulimit -u
16384
```

Many RedHat systems ship with a limit of 1024:

```
$ cat /etc/security/limits.d/90-nproc.conf
# Default limit for number of user's processes to prevent
# accidental fork bombs.
# See rhbz #432903 for reasoning.

*          -    nproc     1024
```

This 1024 should be raised if you errors like the following:

```
EAGAIN The system lacked the  necessary  resources  to  create  another
thread,  or  the  system-imposed  limit  on  the total number of
threads in a process {PTHREAD_THREADS_MAX} would be exceeded.
```

# Multi-node

An TIBCO ActiveSpaces® Transactions application can be, and often is, run on a single node. With High-availability and Distribution features, TIBCO ActiveSpaces® Transactions can run distributed applications across multiple nodes. From an operational point of view, there are very few benefits from running multiple nodes on a single machine. This document recommends and assumes that each node will be run on its own machine.

When an application reaches its throughput limit on a single node, additional performance can be gained by adding multiple nodes. This is called horizontal scaling. For an application that is not designed to be distributed, this often poses a problem. Sometimes this can be addressed by adding a routing device outside of the nodes. But sometimes this cannot be addressed without rewriting the application.

A distributed TIBCO ActiveSpaces® Transactions application can be spread across an arbitrary number of nodes at the High-availability data partition boundary. If the active node for a set of partitions has reached throughput saturation, one or more of the partitions may be migrated to other nodes.

# Analyzing Deadlocks

When TIBCO ActiveSpaces® Transactions detects a deadlock a detailed trace is sent to the log files showing which resource deadlocked, which transactions were involved in the deadlock, which resources they had locked, and which resource they were blocked waiting for. Additionally a stack trace is logged showing where in the application the deadlock occurred.

## Single Node Deadlocks

**Lock order deadlock.** The program below will generate a single transaction lock ordering deadlock between two threads, running in a single JVM, in a single node.

```
// $Revision: 1.1.2.1 $
package com.kabira.snippets.tuning;

import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;

/**
 * Deadlock Example from the ActiveSpaces Transactions Tuning Guide.
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class Deadlock
{

    static MyManagedObject object1;
    static MyManagedObject object2;
```

```
    public static void main(String[] args) throws InterruptedException
    {
        //
        // Create a pair of Managed objects.
        //
        new Transaction("Create Objects")
        {

            @Override
            public void run()
            {
                object1 = new MyManagedObject();
                object2 = new MyManagedObject();
            }
        }.execute();

        //
        // Create a pair of transaction classes to lock them.
        // Giving the object parameters in reverse order will
        // cause two different locking orders, resulting in a deadlock.
        //
        Deadlocker deadlocker1 = new Deadlocker(object1, object2);
        Deadlocker deadlocker2 = new Deadlocker(object2, object1);

        //
        // Run them in separate threads until a deadlock is seen.
        //
        while ((deadlocker1.getNumberDeadlocks() == 0)
            && (deadlocker2.getNumberDeadlocks() == 0))
        {
            MyThread thread1 = new MyThread(deadlocker1);
            MyThread thread2 = new MyThread(deadlocker2);

            thread1.start();
            thread2.start();

            thread1.join();
            thread2.join();
        }
    }

    @Managed
    static class MyManagedObject
    {

        int value;
    }

    static class MyThread extends Thread
    {

        private Deadlocker m_deadlocker;

        MyThread(Deadlocker deadlocker)
        {
            m_deadlocker = deadlocker;
        }

        @Override
        public void run()
        {
            m_deadlocker.execute();
        }
    }

    static class Deadlocker extends Transaction
    {
```

```
        private final MyManagedObject m_object1;
        private final MyManagedObject m_object2;

        Deadlocker(MyManagedObject object1, MyManagedObject object2)
        {
            m_object1 = object1;
            m_object2 = object2;
        }

        @Override
        public void run()
        {

            //
            // This will take a transaction read lock on the first object.
            //
            int value = m_object1.value;

            //
            // Wait a while to maximize the possibility of contention.
            //
            blockForAMoment();

            //
            // This will take a transaction write lock on the second object.
            //
            m_object2.value = 42;

            //
            // Wait a while to maximize the possibility of contention.
            //
            blockForAMoment();
        }

        private void blockForAMoment()
        {
            try
            {
                Thread.sleep(500);
            }
            catch (InterruptedException ex)
            {
            }
        }
    }
}
```

The log file for the A node will contain a trace similar to the following:

```
2010-05-19 10:58:36.190955|OBJ|WARN |19821|osproxy.cpp(857)|deadlock detected in
transaction id 156:2

Transaction 156:2 deadlocked attempting to write lock
  com.kabira.snippets.tuning.Deadlock$MyManagedObject:1 (890258:6145144:112456:1 offset
 64534360)
     locks read { 155:2 }
  which is held by transaction id 155:2

  Callstack for transaction 156:2:
                deadlock on com.kabira.snippets.tuning.Deadlock$MyManagedObject:1

  Objects currently locked in transaction id 156:2
    com.kabira.snippets.tuning.Deadlock$MyManagedObject:2 (890258:6145144:112456:2 offset
 64099448) read lock
```

```
Transaction 155:2 is blocked waiting for a write lock on
  com.kabira.snippets.tuning.Deadlock$MyManagedObject:2 (890258:6145144:112456:2 offset
 64099448)
     locks read { 156:2 }
  which is held by transaction 156:2


  Objects currently locked in transaction 155:2
    com.kabira.snippets.tuning.Deadlock$MyManagedObject:1 (890258:6145144:112456:1 offset
 64534360) read lock

2010-05-19 10:58:36.192736|JAV|WARN |19821|native_tran.cpp(373)|Deadlock detected in Java
 thread:
   com.kabira.ktvm.transaction.DeadlockError:
 at com.kabira.platform.ManagedObject.setInteger(Native Method)
 at com.kabira.snippets.tuning.Deadlock$Deadlocker.run(Deadlock.java:95)
 at com.kabira.platform.Transaction.execute(Transaction.java:310)
 at com.kabira.snippets.tuning.Deadlock$MyThread.run(Deadlock.java:65)
```

Looking at this section by section we see:

```
2010-05-19 10:58:36.190955|OBJ|WARN |19821|osproxy.cpp(857)|deadlock detected in
transaction id 156:2
```

This line is the beginning of the deadlock trace, and shows which transaction id (156:2) detected the deadlock. The transaction which detects the deadlock rolls back and retries.

```
Transaction 156:2 deadlocked attempting to write lock
  com.kabira.snippets.tuning.Deadlock$MyManagedObject:1 (890258:6145144:112456:1 offset
 64534360)
     locks read { 155:2 }
  which is held by transaction id 155:2

  Callstack for transaction 156:2:
                deadlock on com.kabira.snippets.tuning.Deadlock$MyManagedObject:1

  Objects currently locked in transaction id 156:2
    com.kabira.snippets.tuning.Deadlock$MyManagedObject:2 (890258:6145144:112456:2 offset
 64099448) read lock
```

This section shows the transaction which detected the deadlock. What type of lock it was trying to take, which resource it was trying to lock, what other transaction is holding a lock which caused contributed to this deadlock, all transaction locks that it was already holding at the time.

```
Transaction 155:2 is blocked waiting for a write lock on
  com.kabira.snippets.tuning.Deadlock$MyManagedObject:2 (890258:6145144:112456:2 offset
 64099448)
     locks read { 156:2 }
  which is held by transaction 156:2


  Objects currently locked in transaction 155:2
    com.kabira.snippets.tuning.Deadlock$MyManagedObject:1 (890258:6145144:112456:1 offset
 64534360) read lock
```

This section shows the other involved transaction. What type of lock it is blocked waiting to take, which resource it is trying to lock, and what other transaction locks it is currently holding.

```
2010-05-19 10:58:36.192736|JAV|WARN |19821|native_tran.cpp(373)|Deadlock detected in Java
 thread: com.kabira.ktvm.transaction.DeadlockError:
 at com.kabira.platform.ManagedObject.setInteger(Native Method)
 at com.kabira.snippets.tuning.Deadlock$Deadlocker.run(Deadlock.java:95)
 at com.kabira.platform.Transaction.execute(Transaction.java:310)
 at com.kabira.snippets.tuning.Deadlock$MyThread.run(Deadlock.java:65)
```

This section shows the call stack of the transaction which detected the deadlock.

**Promotion deadlock.** Lock promotion is when a transaction currently holding a read lock on an object attempts to acquire a write lock on the same object (i.e. Promoting the read lock to a write lock). If blocking for this write lock would result in deadlock, it is called a promotion deadlock.

The program below will generate a single promotion deadlock between two threads, running in a single JVM, in a single node.

```
// $Revision: 1.1.2.1 $
package com.kabira.snippets.tuning;

import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;

/**
 * Promotion deadlock Example from the ActiveSpaces Transactions Tuning Guide.
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class PromotionDeadlock
{

    static MyManagedObject targetObject;

    public static void main(String[] args) throws InterruptedException
    {
        //
        // Create a Managed objects.
        //
        new Transaction("Create Objects")
        {
            @Override
            public void run()
            {
                targetObject = new MyManagedObject();
            }
        }.execute();

        //
        // Create a pair of transaction classes that will both
        // promote lock the Managed object, resulting in a
        // promotion deadlock.
        //
        Deadlocker deadlocker1 = new Deadlocker(targetObject);
        Deadlocker deadlocker2 = new Deadlocker(targetObject);

        //
        // Run them in separate threads until a deadlock is seen.
        //
        while ((deadlocker1.getNumberDeadlocks() == 0)
            && (deadlocker2.getNumberDeadlocks() == 0))
        {
            MyThread thread1 = new MyThread(deadlocker1);
            MyThread thread2 = new MyThread(deadlocker2);

            thread1.start();
            thread2.start();

            thread1.join();
            thread2.join();
        }
```

```
    }

    @Managed
    static class MyManagedObject
    {

        int value;
    }

    static class MyThread extends Thread
    {

        private Deadlocker m_deadlocker;

        MyThread(Deadlocker deadlocker)
        {
            m_deadlocker = deadlocker;
        }

        @Override
        public void run()
        {
            m_deadlocker.execute();
        }
    }

    static class Deadlocker extends Transaction
    {

        private final MyManagedObject m_targetObject;

        Deadlocker(MyManagedObject targetObject)
        {
            m_targetObject = targetObject;
        }

        @Override
        public void run()
        {

            //
            // This will take a transaction read lock on the object.
            //
            int value = m_targetObject.value;

            //
            // Wait a while to maximize the possibility of contention.
            //
            blockForAMoment();

            //
            // This will take a transaction write lock on the object
            // (promoting the read lock).
            //
            m_targetObject.value = 42;

            //
            // Wait a while to maximize the possibility of contention.
            //
            blockForAMoment();
        }

        private void blockForAMoment()
        {
            try
            {
                Thread.sleep(500);
```

```
            }
            catch (InterruptedException ex)
            {
            }
        }
    }
}
```

The trace messages are similar to those show in the previous section for a lock order deadlock, with the difference being that promotion deadlock will be mentioned:

```
06-01 08:45:57|runtime::Events::TraceWarning|5391|103:177|Warning|promotion deadlock
detected in transaction id 130:2

Transaction 130:2 deadlocked attempting to promote (write lock)
  com.kabira.snippets.tuning.PromotionDeadlock$MyManagedObject:1 (890258:9384688:2001:1
 offset 64698248)
     locks read { 133:2, 130:2 }, promote waiter { 133:2 }
  which is held by transaction id 133:2

  Callstack for transaction 130:2:
                promotion deadlock on
com.kabira.snippets.tuning.PromotionDeadlock$MyManagedObject:1

  Objects currently locked in transaction id 130:2
    com.kabira.snippets.tuning.PromotionDeadlock$MyManagedObject:1 (890258:9384688:2001:1
 offset 64698248) read lock

Transaction 133:2 is blocked waiting for a promote lock on
  com.kabira.snippets.tuning.PromotionDeadlock$MyManagedObject:1 (890258:9384688:2001:1
 offset 64698248)
     locks read { 133:2, 130:2 }, promote waiter { 133:2 }
  which is held by transaction 130:2

  Objects currently locked in transaction 133:2
    com.kabira.snippets.tuning.PromotionDeadlock$MyManagedObject:1 (890258:9384688:2001:1
 offset 64698248) read lock

06-01 08:45:57|runtime::Events::TraceWarning|5391|103:179|Warning|Deadlock detected in
Java thread: com.kabira.ktvm.transaction.DeadlockError:
 at com.kabira.platform.ManagedObject.setInteger(Native Method)
 at com.kabira.snippets.tuning.PromotionDeadlock$Deadlocker.run(PromotionDeadlock.java:92)

 at com.kabira.platform.Transaction.execute(Transaction.java:309)
 at com.kabira.snippets.tuning.PromotionDeadlock$MyThread.run(PromotionDeadlock.java:63)
```

**Complex deadlock.** The previous examples showed simple deadlocks, occurring between two transactions. More complex deadlocks are possible involving more than two transactions. For example, transaction 1 deadlocks trying to acquire a lock on an object held by transaction 2 who is blocked waiting on a object held by transaction 3.

To aid in analyzing complex deadlocks the following will be found in the trace messages:

For each contended object, a display of the locks is included, including any promotion waiters.

If the runtime detects that a deadlock happens due to a read lock being blocked, it includes the transaction blocked waiting for the promotion.

# Distributed deadlocks

Single node deadlocks are bad for performance because they are a source of contention, leading to lower throughput, higher latency and higher CPU cost. But the deadlocks are detected immediately, because each node has a built in transaction lock manager.

Distributed deadlocks are **extremely** bad for performance because they use a timeout mechanism for deadlock detection. The default setting for this timeout is 60 seconds in a production build.

The program below will generate a distributed transaction lock ordering deadlock between two transactions running across multiple nodes.

```
// $Revision: 1.1.2.5 $
package com.kabira.snippets.tuning;

import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;
import com.kabira.platform.highavailability.PartitionManager;
import com.kabira.platform.highavailability.PartitionManager.EnableAction;
import com.kabira.platform.highavailability.PartitionMapper;
import com.kabira.platform.highavailability.ReplicaNode;
import static com.kabira.platform.highavailability.ReplicaNode.ReplicationType.*;
import com.kabira.platform.property.Status;

/**
 * Distributed deadlock example from the ActiveSpaces Transactions Tuning Guide
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainname</b> = Development
 * </ul>
 * Note this sample blocks on B and C nodes,
 * and needs to be explicitly stopped.
 */
public class DistributedDeadlock
{
    private static TestObject object1;
    private static TestObject object2;
    static final String nodeName = System.getProperty(Status.NODE_NAME);

    public static void main(String[] args) throws InterruptedException
    {
        //
        // Block all but the A node.
        //
        new NodeChecker().blockAllButA();

        //
        // Define the partitions to be used by this snippet
        //
        new PartitionCreator().createPartitions();

        //
        // Create a pair of objects, one active on node B,
        // and the other active on node C.
        //
        new Transaction("Create Objects")
        {
            @Override
            public void run()
            {
                object1 = new TestObject();
                object2 = new TestObject();
```

```
                //
                // For each distributed object, assign it a
                // reference to the other.
                //
                object1.otherObject = object2;
                object2.otherObject = object1;
            }
        }.execute();

        //
        // Create a pair of objects, one active on node B,
        // and the other active on node C.
        //
        new Transaction("Spawn Deadlockers")
        {
            @Override
            public void run()
            {
                //
                // Ask them each to spawn a Deadlocker thread.
                // This should execute on node B for one of them
                // and node C for the other.
                //
                object1.spawnDeadlocker();
                object2.spawnDeadlocker();
            }
        }.execute();

        //
        // Now block main in the A node to keep the JVM from exiting.
        //
        new NodeChecker().block();
    }

    private static class PartitionCreator
    {
        void createPartitions()
        {
            new Transaction("Partition Definition")
            {
                @Override
                protected void run() throws Rollback
                {
                    //
                    //  Set up the node lists - notice that the odd node list
                    //  has node B as the active node, while the even
                    //  node list has node C as the active node.
                    //
                    ReplicaNode [] evenReplicaList = new ReplicaNode []
                    {
                        new ReplicaNode("C", SYNCHRONOUS),
                        new ReplicaNode("A", SYNCHRONOUS)
                    };
                    ReplicaNode [] oddReplicaList = new ReplicaNode []
                    {
                        new ReplicaNode("B", SYNCHRONOUS),
                        new ReplicaNode("A", SYNCHRONOUS)
                    };

                    //
                    //  Define two partitions
                    //
                   PartitionManager.definePartition("Even", null, "B", evenReplicaList);

                    PartitionManager.definePartition("Odd", null, "C", oddReplicaList);

                    //
```

```
                //  Enable the partitions
                //
                PartitionManager.enablePartitions(
                        EnableAction.JOIN_CLUSTER_PURGE);

                //
                //  Install the partition mapper
                //
                PartitionManager.setMapper(
                        TestObject.class, new AssignPartitions());
            }
        }.execute();
    }
}

//
//  Partition mapper that maps objects to either Even or Odd
//
private static class AssignPartitions extends PartitionMapper
{

    @Override
    public String getPartition(Object obj)
    {
        this.m_count++;
        String partition = "Even";

        if ((this.m_count % 2) == 1)
        {
            partition = "Odd";
        }

        return partition;
    }
    private Integer m_count = 0;
}

@Managed
private static class TestObject
{
    TestObject  otherObject;
    String      data;

    public void lockObjects()
    {
        Transaction.setTransactionDescription("locking first object");
        this.doWork();

        //
        // Delay longer on the B node to try to force the deadlock
        // to occur on the C.  Otherwise, both sides could see
        // deadlocks at the same time, making the log files less clear
        // for this snippet.
        //
        if (nodeName.equals("B"))
        {
            block(10000);
        }
        else
        {
            block(500);
        }

        Transaction.setTransactionDescription("locking second object");
        otherObject.doWork();

        block(500);
```

```
        }

        public void spawnDeadlocker()
        {
            new DeadlockThread(this).start();
        }

        private void block(int milliseconds)
        {
            try
            {
                Thread.sleep(milliseconds);
            }
            catch (InterruptedException ex)
            {
            }
        }

        private void doWork()
        {
            data = "work";
        }
    }

    private static class DeadlockThread extends Thread
    {

        private Transaction m_deadlockTransaction;

        DeadlockThread(TestObject object)
        {
            m_deadlockTransaction =
                new DeadlockTransaction("DeadlockThread", object);
        }

        @Override
        public void run()
        {
            while (true)
            {
                if (m_deadlockTransaction.execute()
                        == Transaction.Result.ROLLBACK)
                {
                    return;
                }
            }
        }
    }

    private static class DeadlockTransaction extends Transaction
    {

        private final TestObject m_object;

        DeadlockTransaction(final String name, TestObject object)
        {
            super(name);
            m_object = object;
        }

        @Override
        public void run() throws Rollback
        {
            if (getNumberDeadlocks() != 0)
            {
                System.out.println("A deadlock has been seen, "
                        + "you may now stop the distributed application");
```

```
                throw new Transaction.Rollback();
            }
            m_object.lockObjects();
        }
    }

    private static class NodeChecker
    {
        //
        // If we are not the A node, block here forever
        //
        void blockAllButA()
        {
            while (!nodeName.equals("A"))
            {
                block();
            }
        }

        public void block()
        {
            while (true)
            {
                try
                {
                    Thread.sleep(500);
                } catch (InterruptedException ex)
                {
                }
            }
        }
    }
}
```

The program should produce a deadlock that is processed on node C, and found in the node C application log file, looking similar to:

```
2014-01-30 12:45:05.462557|ENG|WARN
|5188|engine.cpp(2844)|com.kabira.ktvm.transaction.DeadlockError: Global transaction
serializable:67:142:1:396494485533021 deadlock processed on node C
Objects locked in local transaction 'DeadlockThread'[142:1, tid 5188, locking second
object]
    com.kabira.snippets.tuning.DistributedDeadlock$TestObject:63
(890258:8376920:396485028116479:63) write lock

Blocked transactions on local node:
Global transaction id: serializable:66:139:1:396489640676425
Transaction [139:3, tid 5153] is blocked waiting for a write lock on
  com.kabira.snippets.tuning.DistributedDeadlock$TestObject:63
(890258:8376920:396485028116479:63)
     locks write { 'DeadlockThread'[142:1, tid 5188, locking second object] }

  Callstack for transaction 139:3:
    dispatch calling [java dispatch] on
com.kabira.snippets.tuning.DistributedDeadlock$TestObject:63

  Objects currently locked in transaction [139:3, tid 5153]
    com.kabira.snippets.tuning.DistributedDeadlock$TestObject:25
(890258:8376920:396485028116479:25) write lock

===== Start deadlock report for remote node B =====
com.kabira.ktvm.transaction.DeadlockError: distributed deadlock detected in transaction
 id [138:2, tid 5151] [engine application::com_kabira_snippets_tuning_DistributedDeadlock2]

Transaction [138:2, tid 5151] deadlocked attempting to write lock
  com.kabira.snippets.tuning.DistributedDeadlock$TestObject:25
```

```
(890258:8376920:396485028116479:25)
     locks write { 'DeadlockThread'[139:1, tid 5189, locking second object] }
  Callstack for transaction 138:2:
    dispatch calling [java dispatch] on
com.kabira.snippets.tuning.DistributedDeadlock$TestObject:25
   distributed deadlock on com.kabira.snippets.tuning.DistributedDeadlock$TestObject:25


  Objects currently locked in transaction id [138:2, tid 5151]
    com.kabira.snippets.tuning.DistributedDeadlock$TestObject:63
(890258:8376920:396485028116479:63) write lock

Object is write locked in transaction 'DeadlockThread'[139:1, tid 5189, locking second
object]
  Objects currently locked in transaction 'DeadlockThread'[139:1, tid 5189, locking second
 object]
    com.kabira.snippets.tuning.DistributedDeadlock$TestObject:25
(890258:8376920:396485028116479:25) write lock

Object has a total of 1 transactions waiting for a write lock

 at com.kabira.platform.ManagedObject.setReference(Native Method)
 at
com.kabira.snippets.tuning.DistributedDeadlock$TestObject.$doWorkImpl(DistributedDeadlock.java:206)

===== End deadlock report for remote node B =====

 at com.kabira.platform.ManagedObject._sendTwoWay(Native Method)
 at com.kabira.platform.ManagedObject.sendTwoWay(ManagedObject.java:655)
 at
com.kabira.snippets.tuning.DistributedDeadlock$TestObject.doWork(DistributedDeadlock.java)

 at
com.kabira.snippets.tuning.DistributedDeadlock$TestObject.$lockObjectsImpl(DistributedDeadlock.java:183)

 at
com.kabira.snippets.tuning.DistributedDeadlock$TestObject.lockObjects(DistributedDeadlock.java)

 at
com.kabira.snippets.tuning.DistributedDeadlock$DeadlockTransaction.run(DistributedDeadlock.java:255)

 at com.kabira.platform.Transaction.execute(Transaction.java:484)
 at com.kabira.platform.Transaction.execute(Transaction.java:542)
```

Looking section by section through the B log file we see

```
2014-01-30 12:45:05.462557|ENG|WARN
|5188|engine.cpp(2844)|com.kabira.ktvm.transaction.DeadlockError: Global transaction
serializable:67:142:1:396494485533021 deadlock processed on node C
Objects locked in local transaction 'DeadlockThread'[142:1, tid 5188, locking second
object]
    com.kabira.snippets.tuning.DistributedDeadlock$TestObject:63
(890258:8376920:396485028116479:63) write lock
```

This shows the global transaction in which the deadlock timeout occurred, including the name of the transaction (if set), the transaction identifier, and the current current setting of the transaction desription (if set).

It then shows a list of objects already locked within the transaction.

```
Blocked transactions on local node:
Global transaction id: serializable:66:139:1:396489640676425
Transaction [139:3, tid 5153] is blocked waiting for a write lock on
```

```
  com.kabira.snippets.tuning.DistributedDeadlock$TestObject:63
(890258:8376920:396485028116479:63)
     locks write { 'DeadlockThread'[142:1, tid 5188, locking second object] }

  Callstack for transaction 139:3:
    dispatch calling [java dispatch] on
com.kabira.snippets.tuning.DistributedDeadlock$TestObject:63

  Objects currently locked in transaction [139:3, tid 5153]
    com.kabira.snippets.tuning.DistributedDeadlock$TestObject:25
(890258:8376920:396485028116479:25) write lock
```

Then a list of local transactions that are blocked on a lock(s) held by the current local transaction is shown. Here, we see that transaction 139:3 is blocked waiting for a write lock on TestObject:63, which is held by the deadlocking transaction, 142:1.

```
===== Start deadlock report for remote node B =====
com.kabira.ktvm.transaction.DeadlockError: distributed deadlock detected in transaction
 id [138:2, tid 5151] [engine application::com_kabira_snippets_tuning_DistributedDeadlock2]

Transaction [138:2, tid 5151] deadlocked attempting to write lock
  com.kabira.snippets.tuning.DistributedDeadlock$TestObject:25
(890258:8376920:396485028116479:25)
     locks write { 'DeadlockThread'[139:1, tid 5189, locking second object] }
  Callstack for transaction 138:2:
    dispatch calling [java dispatch] on
com.kabira.snippets.tuning.DistributedDeadlock$TestObject:25
   distributed deadlock on com.kabira.snippets.tuning.DistributedDeadlock$TestObject:25


  Objects currently locked in transaction id [138:2, tid 5151]
    com.kabira.snippets.tuning.DistributedDeadlock$TestObject:63
(890258:8376920:396485028116479:63) write lock

Object is write locked in transaction 'DeadlockThread'[139:1, tid 5189, locking second
object]
  Objects currently locked in transaction 'DeadlockThread'[139:1, tid 5189, locking second
 object]
    com.kabira.snippets.tuning.DistributedDeadlock$TestObject:25
(890258:8376920:396485028116479:25) write lock

Object has a total of 1 transactions waiting for a write lock

 at com.kabira.platform.ManagedObject.setReference(Native Method)
 at
com.kabira.snippets.tuning.DistributedDeadlock$TestObject.$doWorkImpl(DistributedDeadlock.java:206)

===== End deadlock report for remote node B =====
```

Next we see a description of the local transaction on the node where the deadlock occurred. In this case, node B, where local transaction 138:2 deadlocked trying to acquire a write lock on TestObject:25, which was held by transaction 139:1

```
 at com.kabira.platform.ManagedObject._sendTwoWay(Native Method)
 at com.kabira.platform.ManagedObject.sendTwoWay(ManagedObject.java:655)
 at
com.kabira.snippets.tuning.DistributedDeadlock$TestObject.doWork(DistributedDeadlock.java)

 at
com.kabira.snippets.tuning.DistributedDeadlock$TestObject.$lockObjectsImpl(DistributedDeadlock.java:183)

 at
com.kabira.snippets.tuning.DistributedDeadlock$TestObject.lockObjects(DistributedDeadlock.java)
```

```
 at
com.kabira.snippets.tuning.DistributedDeadlock$DeadlockTransaction.run(DistributedDeadlock.java:255)

 at com.kabira.platform.Transaction.execute(Transaction.java:484)
 at com.kabira.platform.Transaction.execute(Transaction.java:542)
```

The last section shows a stack backtrace, including source file names, and line numbers, of where the originating distributed transaction deadlock occurred.

# Analyzing Transaction Lock Contention

The `transaction` statistic can show which classes are involved in transaction lock contention. Often, this is sufficient to help the developer already familiar with the application, identify application changes for reducing the contention. For cases where the code paths involved in the contention are not already known, the `transactioncontention` statistic can be useful.

Enabling the `transactioncontention` statistic causes the TIBCO ActiveSpaces® Transactions runtime to collect a stack backtrace each time a transaction lock encounters contention. The stacks are saved per managed class name.

The collection of transaction contention statistics is very expensive computationally and should only be used in development or test systems.

To use transaction contention statistics, enable them with the `administrator enable statistics statistics=transactioncontention` command.

If your application is not already running, start it. This example uses the TransactionContention snippet shown below.

```
// $Revision: 1.1.2.1 $
package com.kabira.snippets.tuning;

import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;

/**
 * Simple transaction contention generator
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 * Note this sample needs to be explicitly stopped.
 */
public class TransactionContention
{
    public static void main(String[] args)
    {
        //
        // Create a managed object to use for
        // generating transaction lock contention
        //
        final MyManaged myManaged = createMyManaged();

        //
        // Create/start a thread which will
        // transactionally contend for the object.
        //
        new MyThread(myManaged).start();

        while (true)
```

```
        {
            //
            // Contend for the object here
            // from // the main thread (competing
            // with the thread started above).
            //
            generateContention(myManaged);
            nap(200);
        }
    }

    static MyManaged createMyManaged()
    {
        return new Transaction("createMyManaged")
        {
            MyManaged m_object;

            @Override
            protected void run()
            {
                m_object = new MyManaged();
            }

            MyManaged create()
            {
                execute();
                return m_object;
            }
        }.create();
    }

    static void generateContention(final MyManaged myManaged)
    {
        new Transaction("generateContention")
        {
            @Override
            protected void run()
            {
                writeLockObject(myManaged);
            }
        }.execute();
    }

    @Managed
    static class MyManaged
    {
    }

    static void nap(int milliseconds)
    {
        try
        {
            Thread.sleep(milliseconds);
        }
        catch (InterruptedException e)
        {
        }
    }

    static class MyThread extends Thread
    {
        MyManaged m_object;

        MyThread(MyManaged myManaged)
        {
            m_object = myManaged;
        }
```

```
        @Override
        public void run()
        {
            while (true)
            {
                generateContention(m_object);
                nap(200);
            }
        }
    }
}
```

After your application has run long enough to generate some transaction lock contention, stop the data collection with the `administrator disable statistics statistics=transactioncontention` command.

Display the collected data with the `administrator display statistics statistics=transactioncontention` command.

```
======== transaction contention report for A ========

24 occurrences on type com.kabira.snippets.tuning.TransactionContention$MyManaged of
stack:

 com.kabira.platform.Transaction.lockObject(Native Method)
 com.kabira.platform.Transaction.writeLockObject(Transaction.java:706)
 com.kabira.snippets.tuning.TransactionContention$2.run(TransactionContention.java:48)
 com.kabira.platform.Transaction.execute(Transaction.java:484)
 com.kabira.platform.Transaction.execute(Transaction.java:542)

com.kabira.snippets.tuning.TransactionContention.generateContention(TransactionContention.java:43)


com.kabira.snippets.tuning.TransactionContention$MyThread.run(TransactionContention.java:84)

57 occurrences on type com.kabira.snippets.tuning.TransactionContention$MyManaged of
stack:

 com.kabira.platform.Transaction.lockObject(Native Method)
 com.kabira.platform.Transaction.writeLockObject(Transaction.java:706)
 com.kabira.snippets.tuning.TransactionContention$2.run(TransactionContention.java:48)
 com.kabira.platform.Transaction.execute(Transaction.java:484)
 com.kabira.platform.Transaction.execute(Transaction.java:542)

com.kabira.snippets.tuning.TransactionContention.generateContention(TransactionContention.java:43)

 com.kabira.snippets.tuning.TransactionContention.main(TransactionContention.java:16)
 sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
 sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
 java.lang.reflect.Method.invoke(Method.java:483)
 com.intellij.rt.execution.application.AppMain.main(AppMain.java:134)
 sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
 sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
 java.lang.reflect.Method.invoke(Method.java:483)
 com.kabira.platform.MainWrapper.invokeMain(MainWrapper.java:65)
```

This output shows the two call paths which experienced contention.

The collected data may be cleared with the `administrator clear statistics statistics=transactioncontention` command.

# Analyzing Transaction Lock Promotion

Transaction lock promotion can lead to deadlocks. The `transaction` statistic can show which classes are involved in transaction lock promotion. Often, this is sufficient to help the developer already familiar with the application, identify application changes for removing the promotion locks. For cases where the code paths involved in the contention are not already known, the `transactionpromotion` statistic can be useful.

Enabling the `transactionpromotion` statistic causes the TIBCO ActiveSpaces® Transactions runtime to collect a stack backtrace each time a transaction lock is promoted from read to write. The stacks are saved per managed class name.

The collection of transaction promotion statistics is very expensive computationally and should only be used in development or test systems.

To use transaction promotion statistics, enable them with the `administrator enable statistics statistics=transactionpromotion` command.

If your application is not already running, start it. This example uses the TransactionPromotion snippet shown below.

```
// $Revision: 1.1.2.1 $
package com.kabira.snippets.tuning;

import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;

/**
 * Simple transaction promotion generator
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class TransactionPromotion
{
    static final MyManaged m_myManaged = createObject();

    public static void main(String[] args)
    {
        new Transaction("promotion")
        {
            @Override
            protected void run()
            {
                readLockObject(m_myManaged);
                // Do promotion
                writeLockObject(m_myManaged);
            }
        }.execute();
    }

    static MyManaged createObject()
    {
        return new Transaction("createObject")
        {
            MyManaged m_object;

            @Override
            protected void run()
```

```
            {
                m_object = new MyManaged();
            }

            MyManaged create()
            {
                execute();
                return m_object;
            }

        }.create();
    }

    @Managed
    static class MyManaged
    {
    }
}
```

After your application has run stop the data collection with the `administrator disable statistics statistics=transactionpromotion` command.

Display the collected data with the `administrator display statistics statistics=transactionpromotion` command.

```
======== Transaction Promotion report for A ========

Data gathered between 2015-03-20 10:27:18 PDT and 2015-03-20 10:28:04 PDT.

1 occurrence on type com.kabira.snippets.tuning.TransactionPromotion$MyManaged of stack:


 com.kabira.platform.Transaction.lockObject(Native Method)
 com.kabira.platform.Transaction.writeLockObject(Transaction.java:706)
 com.kabira.snippets.tuning.TransactionPromotion$1.run(TransactionPromotion.java:29)
 com.kabira.platform.Transaction.execute(Transaction.java:484)
 com.kabira.platform.Transaction.execute(Transaction.java:542)
 com.kabira.snippets.tuning.TransactionPromotion.main(TransactionPromotion.java:22)
 sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
 sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
 java.lang.reflect.Method.invoke(Method.java:483)
 com.intellij.rt.execution.application.AppMain.main(AppMain.java:134)
 sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
 sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
 java.lang.reflect.Method.invoke(Method.java:483)
 com.kabira.platform.MainWrapper.invokeMain(MainWrapper.java:65)
```

This output shows the two call path where the promotion occurred.

The collected data may be cleared with the `administrator clear statistics statistics=transactionpromotion` command.

# 5

# Performance monitoring

This chapter describes the tools and approach to monitoring TIBCO ActiveSpaces® Transactions applications.

## JVM Tools

Standard Java tools may be used to monitor the JVM.

### Visual VM

https://visualvm.dev.java.net

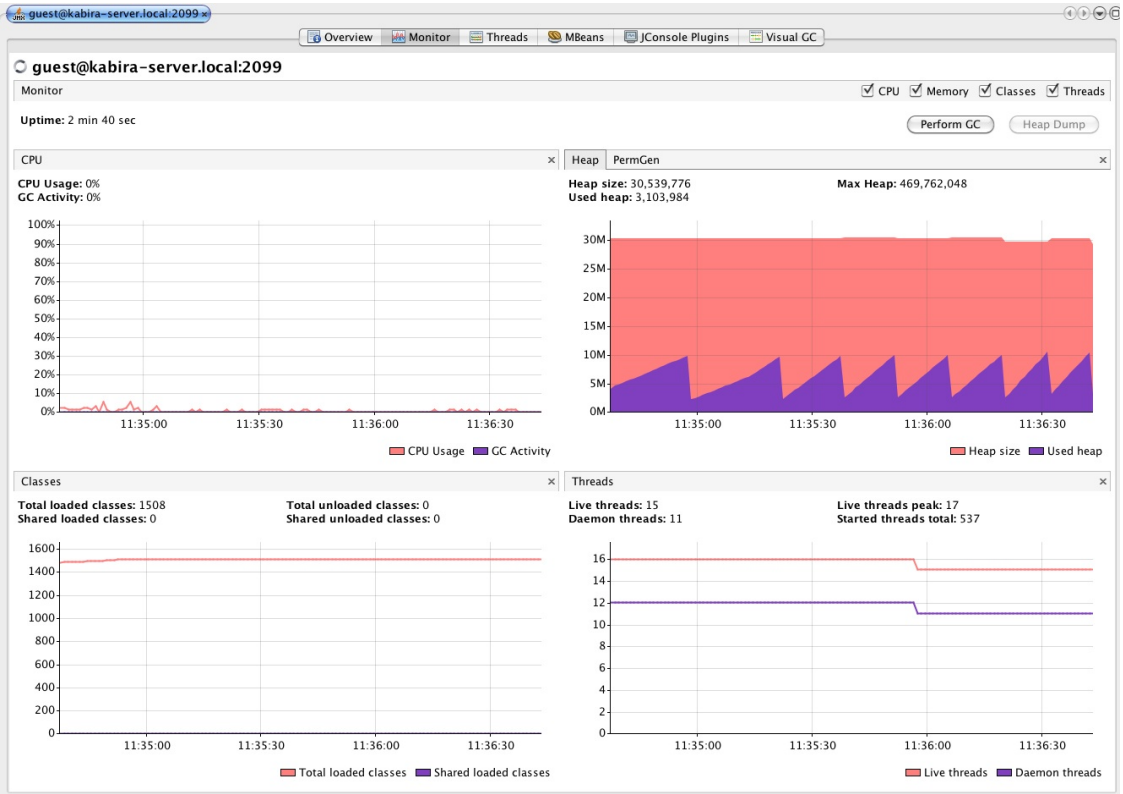



**Figure 5.1. Visual VM**

# JConsole

http://openjdk.java.net/tools/svc/jconsole

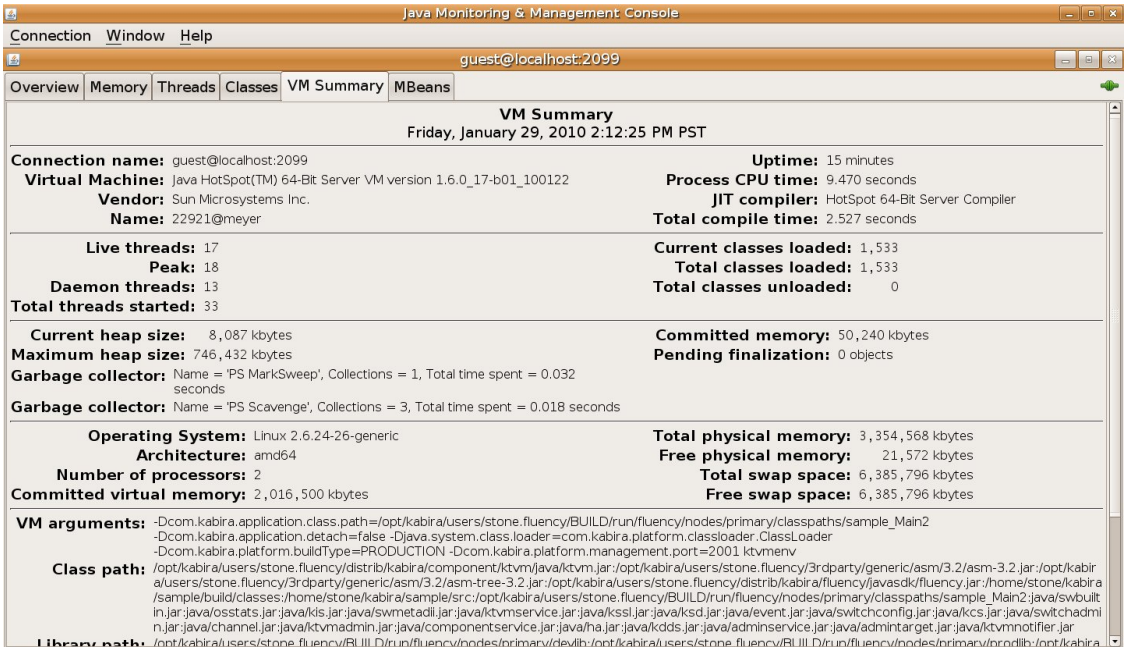



**Figure 5.2. JConsole**

# Java Mission Control and Flight Recorder

Flight Recorder - If using Oracle Java 7 or later, Java Mission Control and Flight Recorder is a built-in, low overhead tool for collecting JVM diagnostic and profiling data.

To enable (but not start) add the following VM deploy options:

```
-XX:+UnlockCommercialFeatures -XX:+FlightRecorder
```

The related GUI tool is Java Mission Control, jmc. Once the application is started your test run, you can select your JVM in mission control, select flight recorder and start recording.

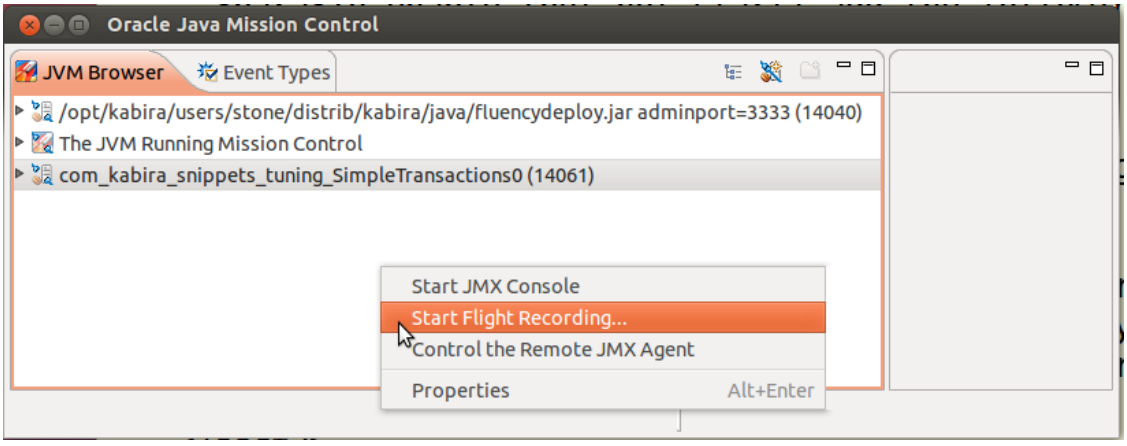



**Figure 5.3. Java Mission Control, JVM Select**

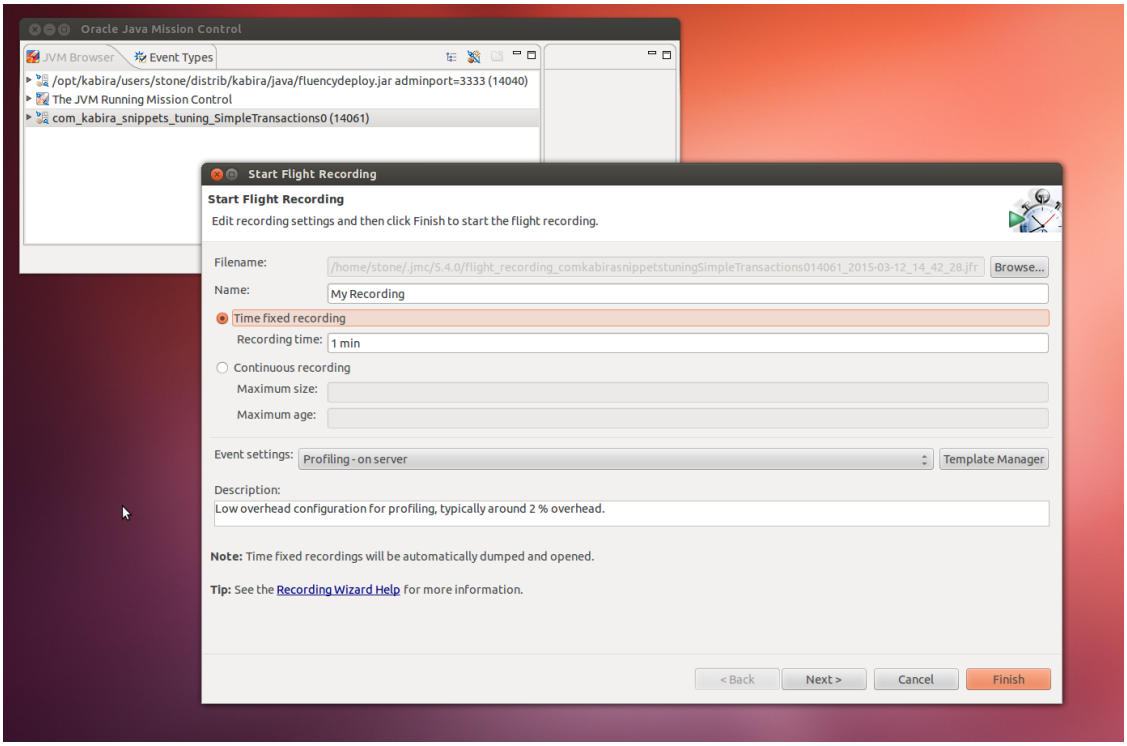



**Figure 5.4. Start Flight Recorder**

Once the capture is finished, mission control will enable exploring the captured data.

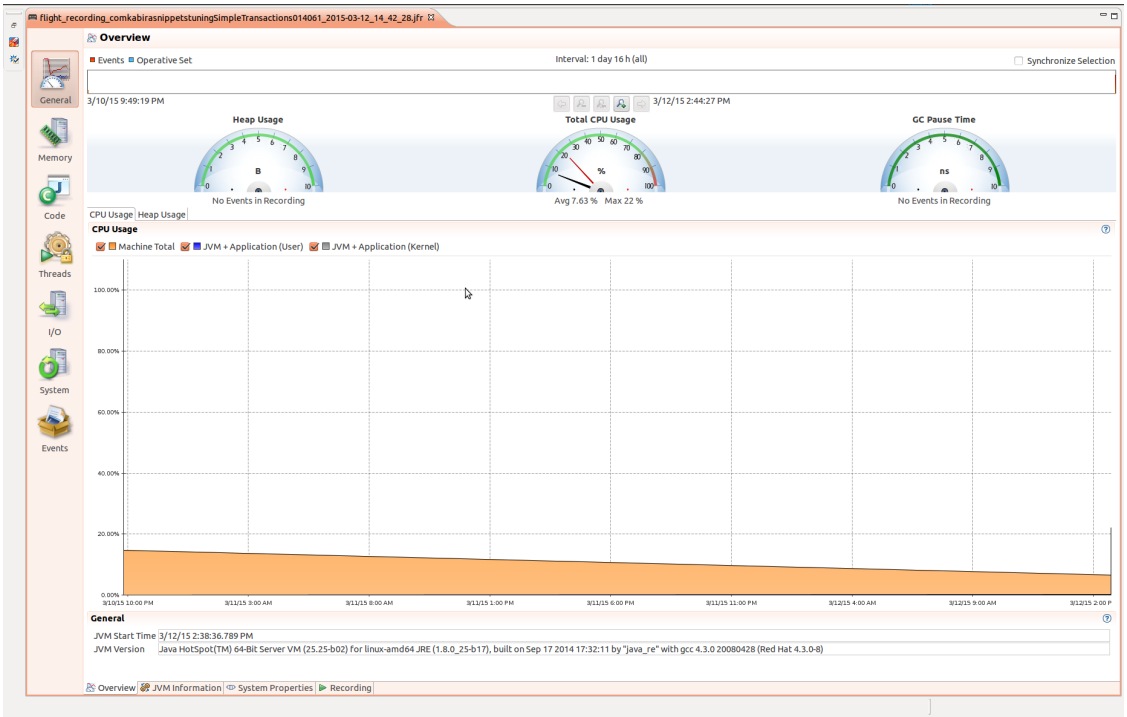



**Figure 5.5. Java Mission Control Explorer**

Recording from the command line my be done with the `jcmd` command. See the jcmd manual page from the JDK, or Oracle online documentation for details.

# Graphical monitoring tools

Graphical display of application transaction statistics, application business state machines, CPU utilization, and shared memory utilization is available from the Node panel of TIBCO ActiveSpaces® Transactions Administrator. Click the `Start monitors` button:

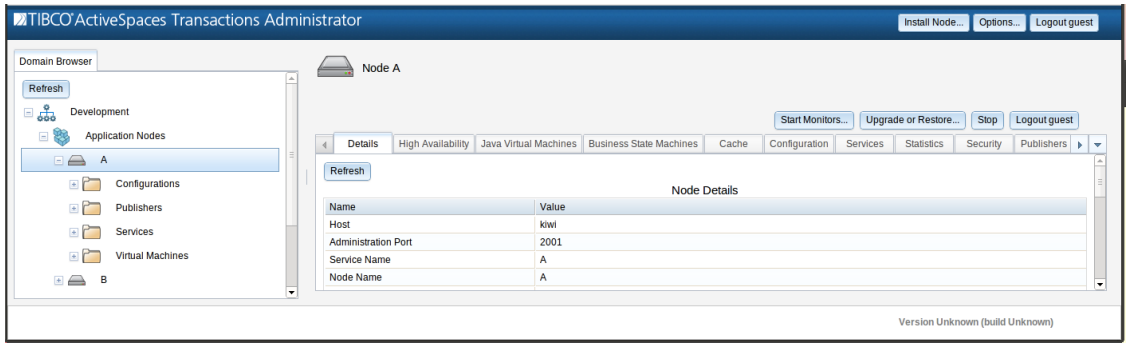



**Figure 5.6. Starting system monitors**

This will open window which allows the selection of which graphical monitors to start.

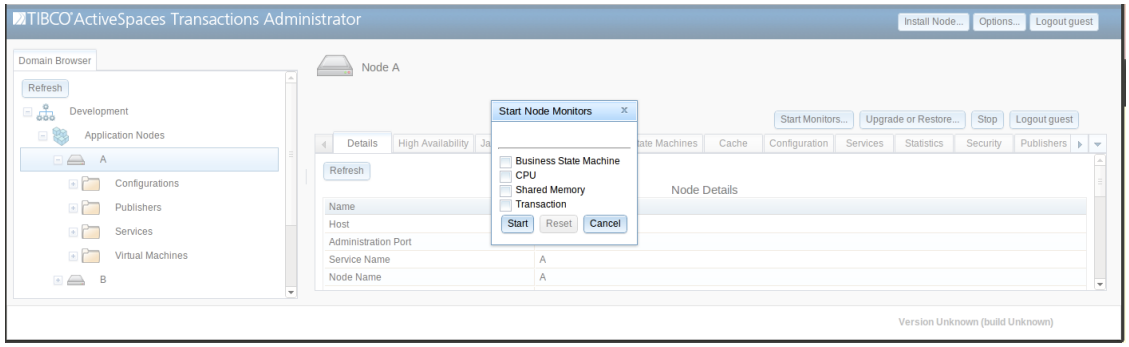



**Figure 5.7. Select system monitors to start**

# Transactions

The runtime continually collects a node-wide view of the aggregate transaction rate per second, and the average execution time. The Transaction Monitor in TIBCO ActiveSpaces® Transactions Administrator is used to display both the transaction rate and the average transaction execution time.

The transaction rate is a count of all transactions executing over a period of time.

The transaction execution time is an average of the execution time of all transactions over a period of time. The transaction execution time is the total time from when a transaction is started to when it commits or rolls-back. This time includes all, or a sub-set, of the following depending on the transaction:

- application code execution time

- transaction lock acquisition and logging time
- network latency to remote nodes for distributed transactions
- replication latency to remote nodes for highly-available objects
- transaction commit or rollback time

The Transaction Monitor has these controls:

- Time vs. Rate - Display transaction rate or execution time.
- Sample Duration - Set the sample duration in seconds. This controls the interval between querying the server for new data for the display.
- Pause or Start - Pause or re-start transaction monitoring.

Figure 5.8 shows an example of displaying the node transaction rate.

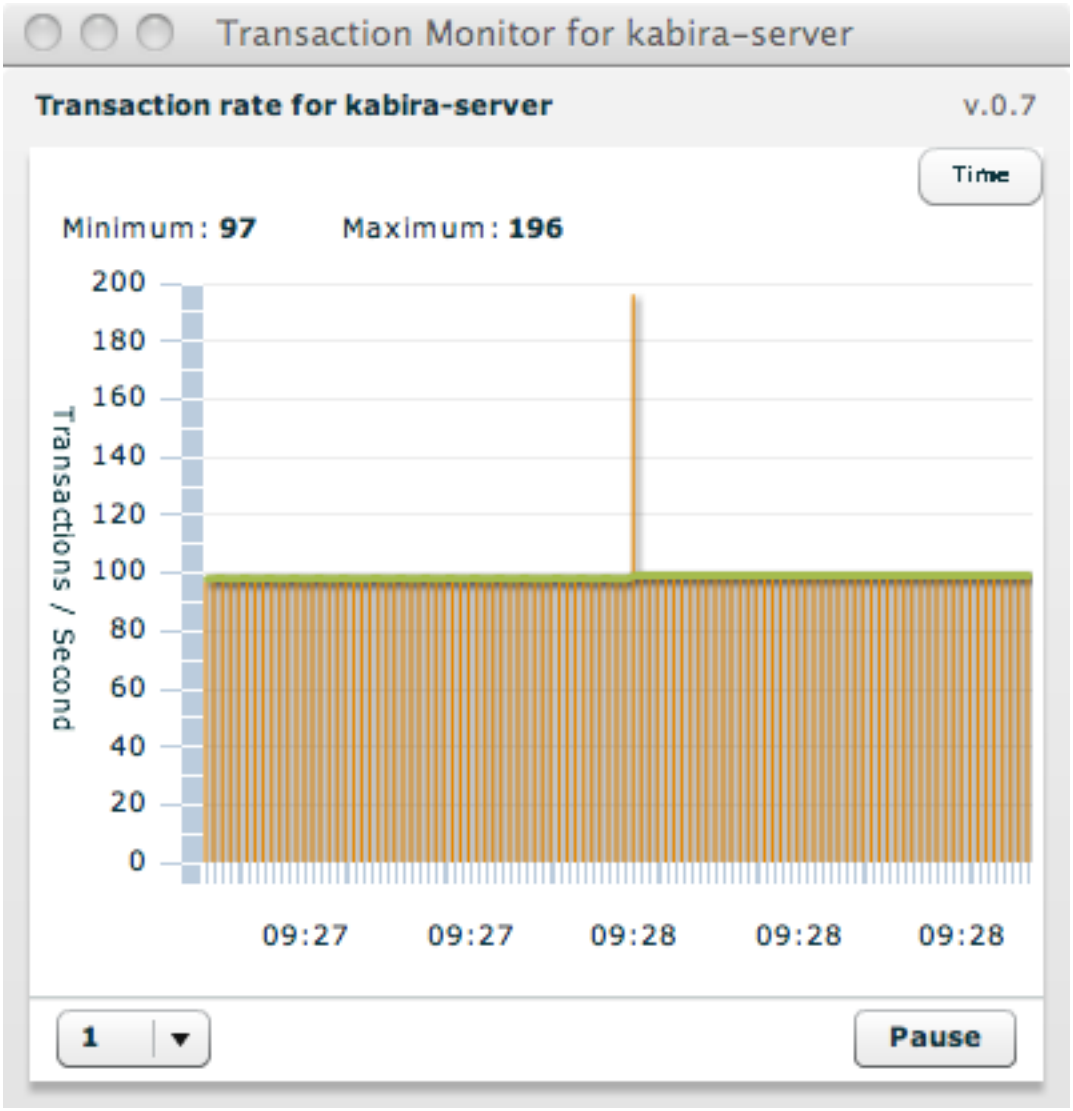



**Figure 5.8. Node transaction rate**

When the Transaction Monitor is started on the domain, the results show the domain-wide aggregate application transaction rate. Figure 5.9 shows an example of displaying the domain-wide transaction rate.

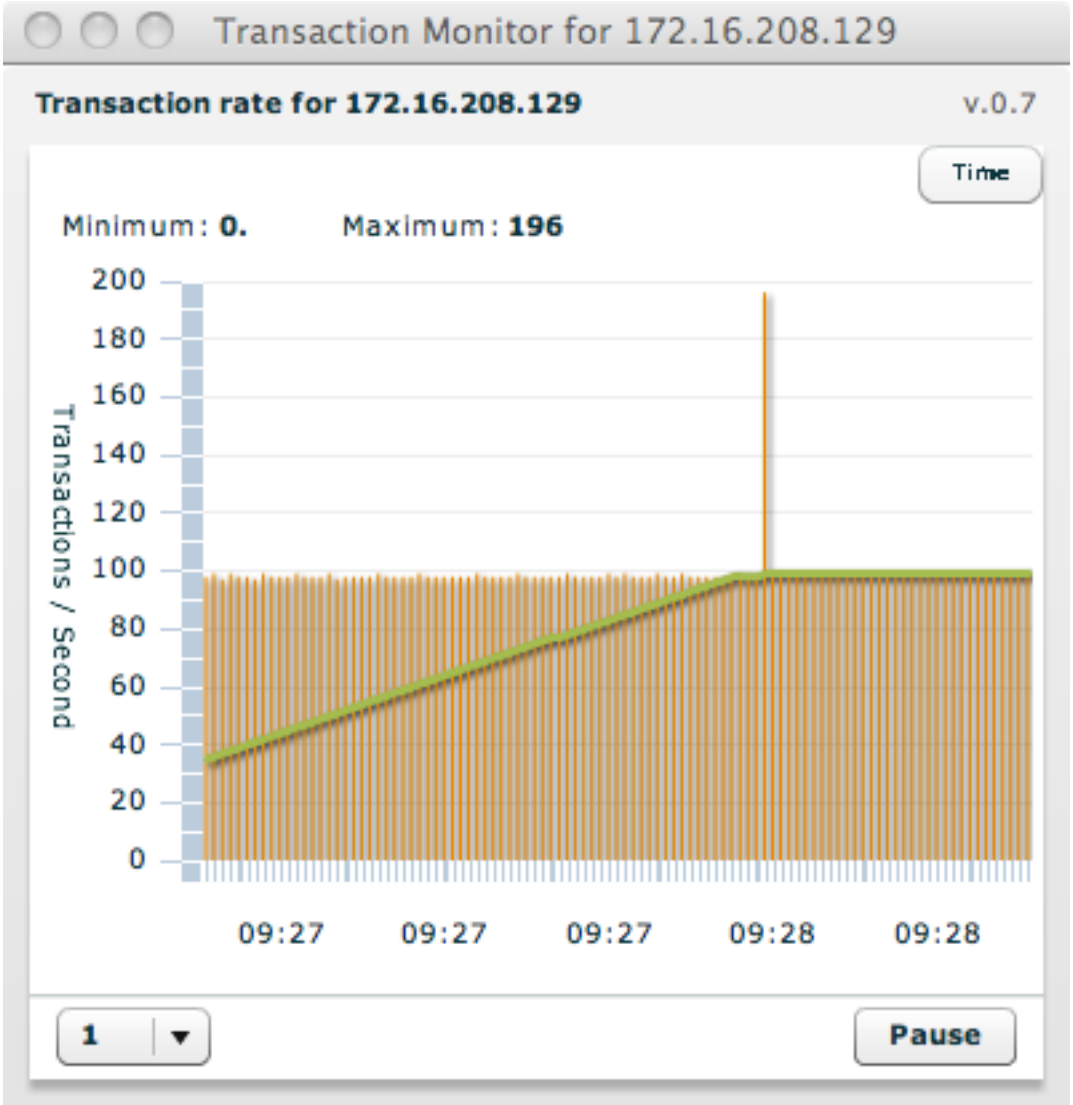



**Figure 5.9. Domain-wide transaction rate**

The Transaction Monitor displays the following information when displaying transaction rates:

- `Network address` - the address of the node or domain manager being monitored.
- `Minimum` - the minimum transaction rate since the monitor was started.
- `Maximum` - the maximum transaction rate since the monitor was started.
- `Transactions / Second` - a graphical display of the number of transactions per second plotted against time.

Selecting `Time` using the Time vs. Rate button on the Transaction Monitor displays the execution time for transactions on a single node or aggregated for all nodes in the domain. Figure 5.10 shows an example of transaction execution time for a node and Figure 5.11 shows the same thing for a domain.

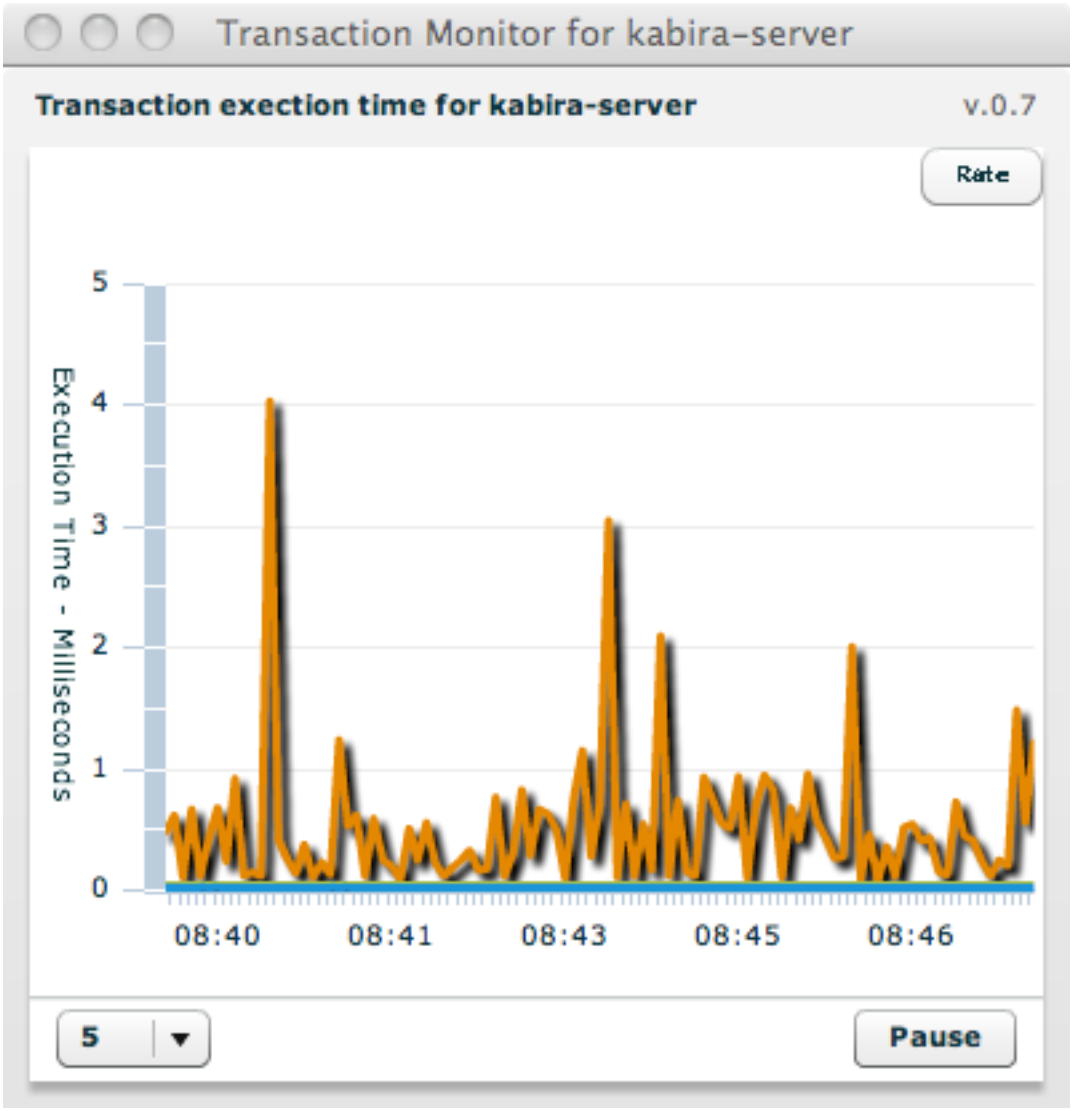



**Figure 5.10. Node transaction execution time**

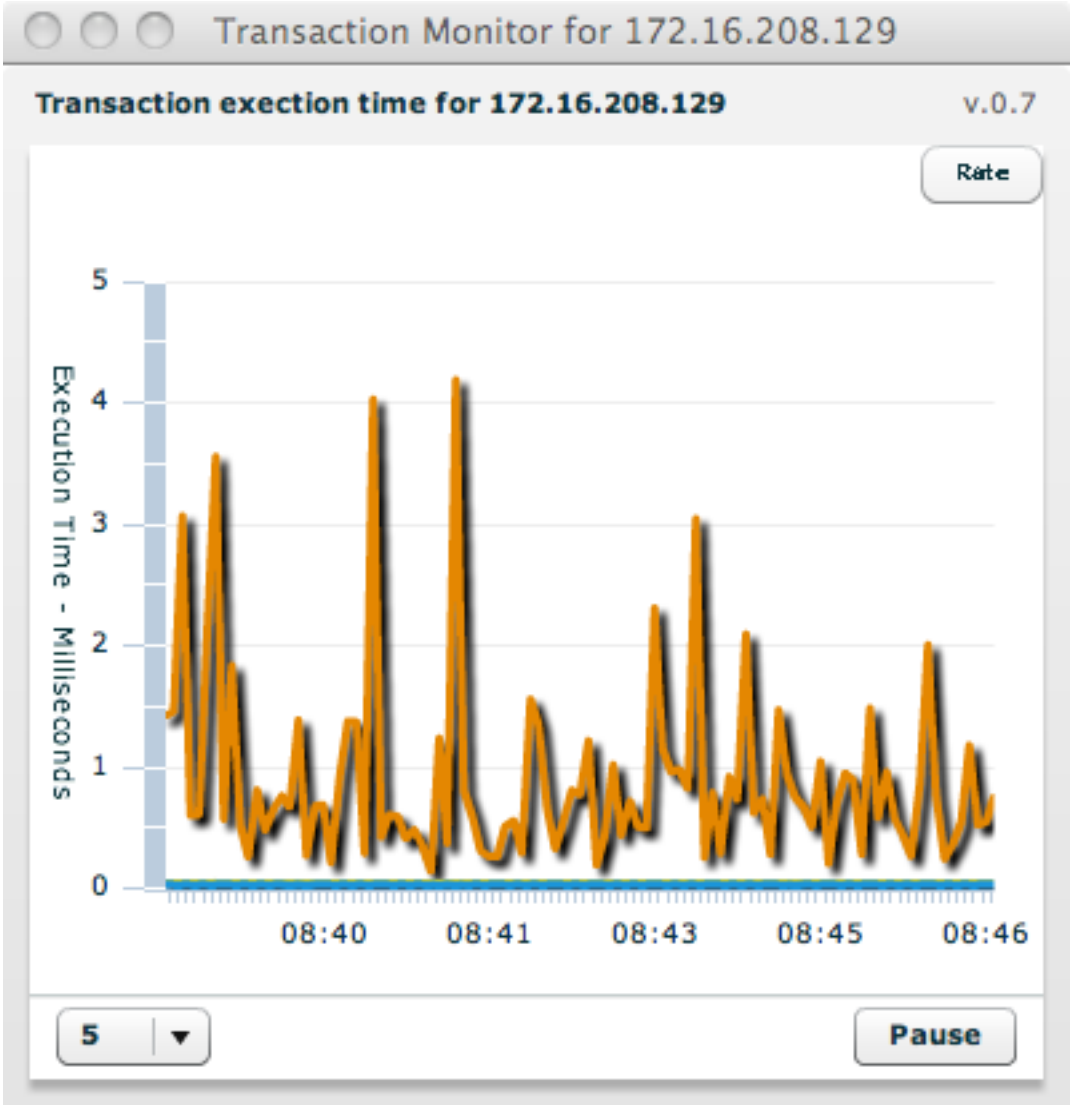



**Figure 5.11. Domain-wide transaction execution time**

The Transaction Monitor displays the following information when displaying transaction execution time:

- `Network address` - the address of the node or domain manager being monitored.
- `Execution Time` - a graphical display of the minimum, maximum, and average execution time in milliseconds plotted against time. The different values are viewed by selecting the monitor and hovering a mouse pointer over the plot.

# Business state machines

Documentation for the graphical business state machine monitor can be found in the administration section of the site documentation for the Business State Machine component:

[http://downloads.fluency.kabira.com/sites/businessstatemachine/administration]

# CPU monitor

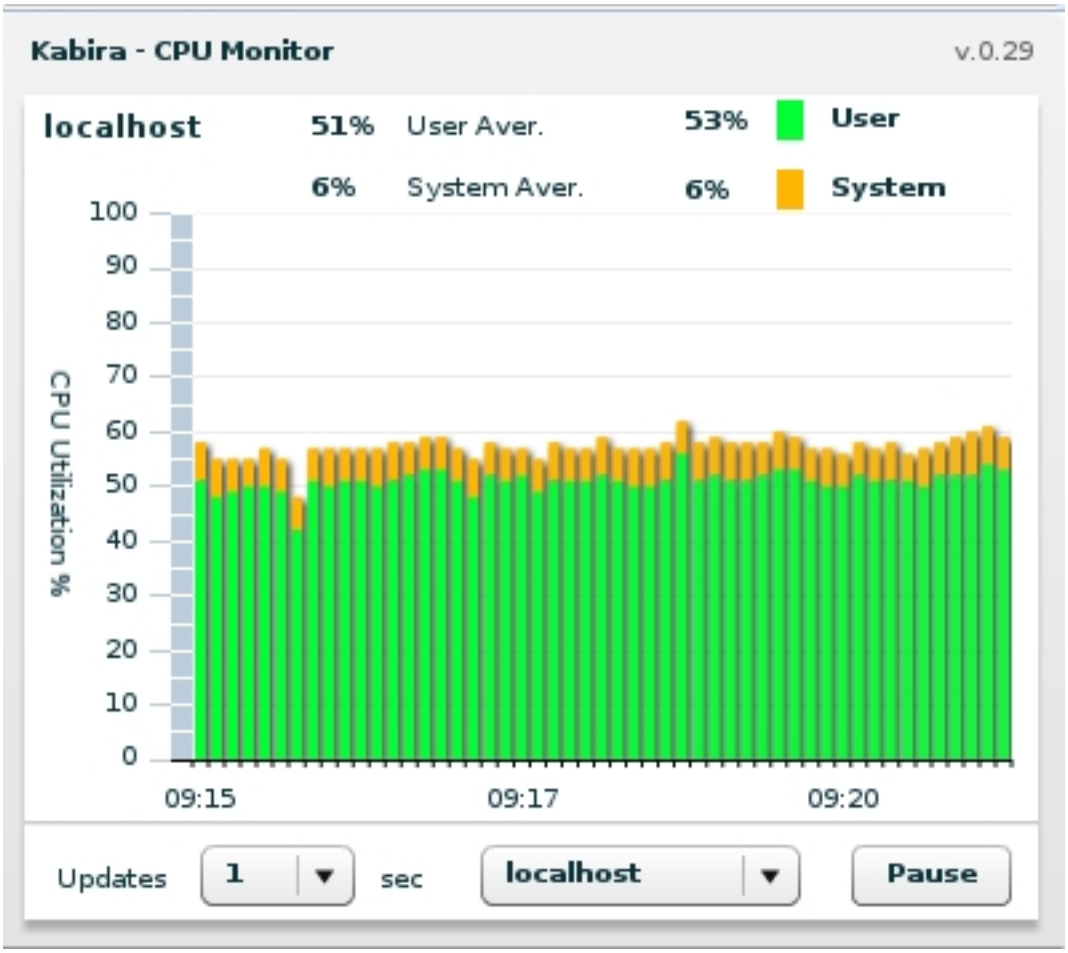



**Figure 5.12. CPU monitor**

## Shared memory monitor

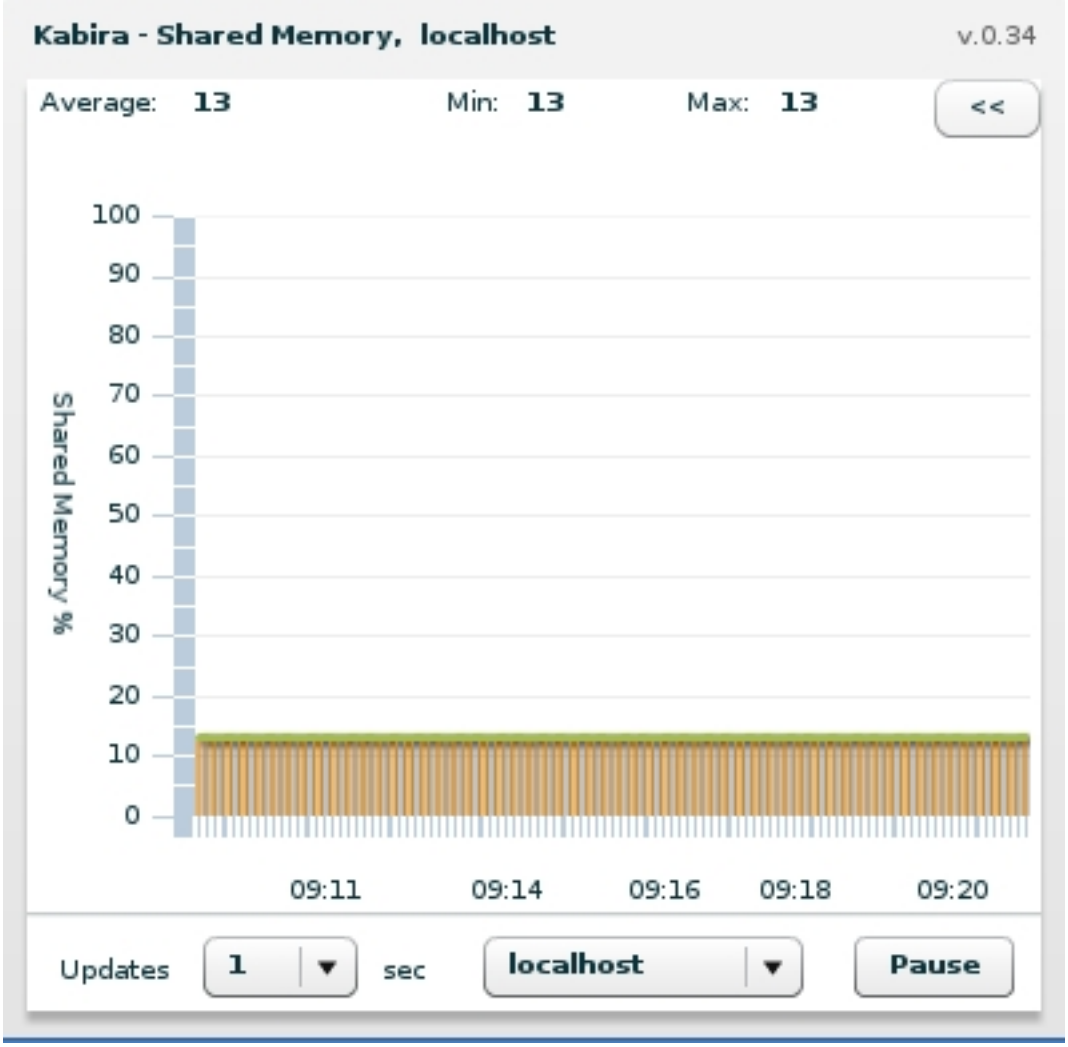



**Figure 5.13. Shared memory monitor**

# The statistics tab

The TIBCO ActiveSpaces® Transactions Administrator offers access a variety of statistics via the `statistics` tab of the `Node` panel. Application, node, and system level reports are available and are selected via the pull down menu:

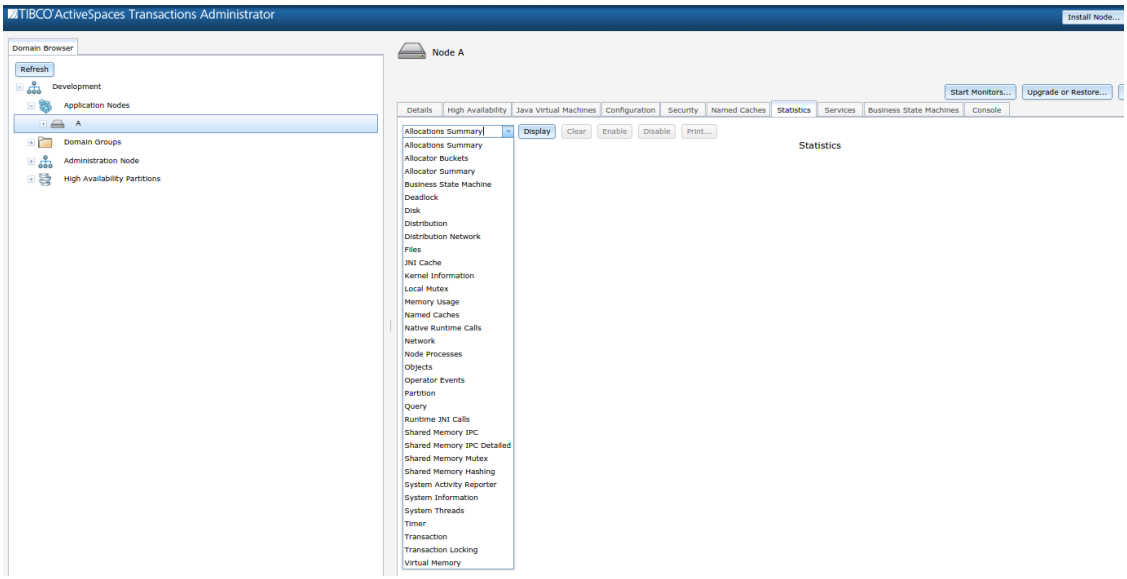



**Figure 5.14. Node statistics menu**

Some of these statistics are collected automatically by the runtime. For these, the `Enable` and `Disable` buttons will not be greyed out and not clickable. Clicking the `Display` button will show the current values.

Other statistics need to be enabled in order to activate data collection. Generally this is because the collection of these statistics imposes some performance, memory usage, or disk usage penalty upon the system. For these, the `Enable` button must be clicked, and the desired amount of time should be waited before clicking the `Display` button to show the collected statistics. It is also good practice to disable the statistic collection, by pressing the `Disable` button, before displaying the report. This restores the system to its previous performance level, and also keeps the reporting itself from showing up in the measurement.

Some statistics support clearing. Those statistics may be cleared at any time by pressing the `Clear` button. For statistics that do not support clearing the `Clear` button will be greyed out and not clickable.

After a report is displayed, it may be printed by clicking the `Print` button.

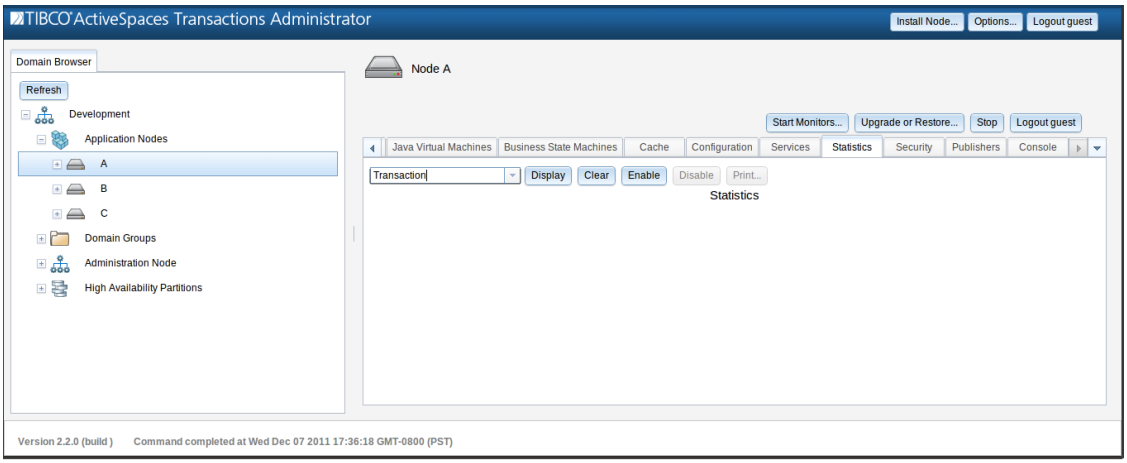
# Application monitoring

## Per Transaction Statistics

Finer grained transaction statistics, tied to each class which implements com.kabira.platform.Transaction are also available. The collection of these statistics imposes a slight performance penalty and is not enabled by default.

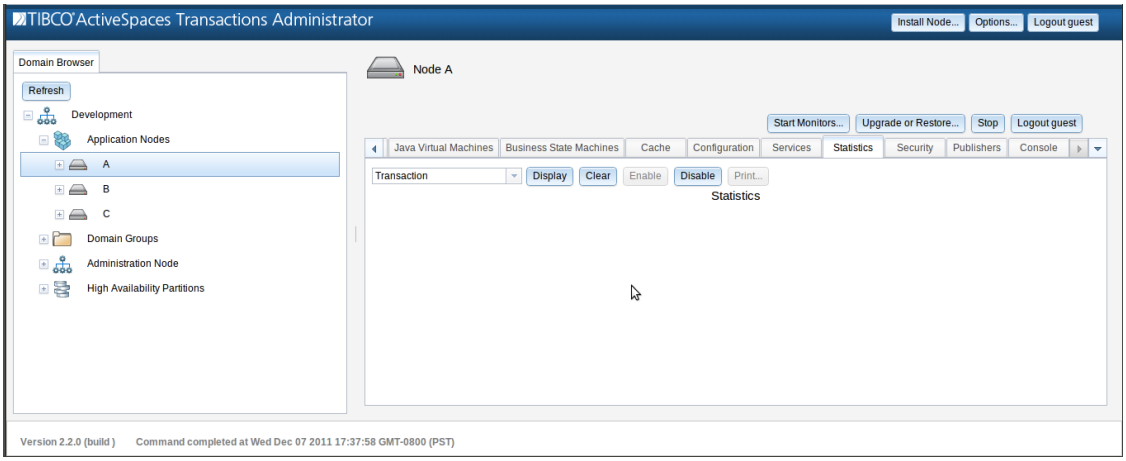
These statistics are available while the node is running.

To collect these statistics, select Transaction in the pull down menu in the statistics panel:




## Figure 5.15. Transaction class statistics

Next enable the collection of these statistics by clicking the `Enable` button:

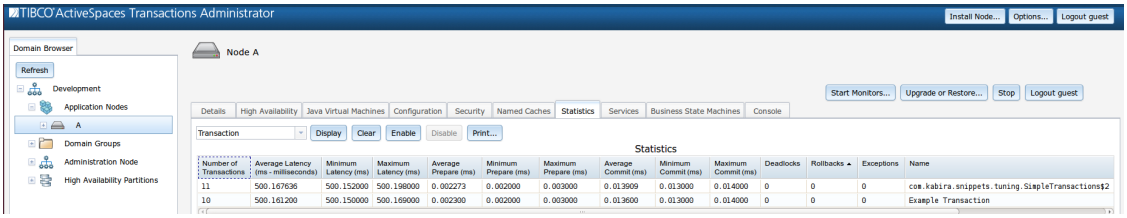



## Figure 5.16. Enable transaction class statistics collection

Allow the data to collect for several seconds or more, and then disable statistics collection by clicking the `Disable` button. Disabling the statistics collection removes the slight performance penalty and allows the system to run at full speed.

Disabling the statistics collection does not remove the collected statistics.

Display the collected statistics by clicking the `Display` button:




## Figure 5.17. Disable and display transaction class statistics

- Number of Transactions - The number of times this transaction executed and committed while the statistic was enabled (exceptions and rollbacks are not counted).
- Average Latency - the average execution time for this transaction. This number includes the prepare and commit times.
- Minimum Latency - the minimum execution time for this transaction. This number includes the prepare and commit times.
- Maximum Latency - the maximum execution time for this transaction. This number includes the prepare and commit times.
- Average Prepare - the average execution time for the prepare phase of this transaction.
- Minimum Prepare - the minimum execution time for the prepare phase of this transaction.
- Maximum Prepare - the maximum execution time for the prepare phase of this transaction.
- Average Commit - the average execution time for the commit phase of this transaction.
- Minimum Commit - the minimum execution time for the commit phase of this transaction.
- Maximum Commit - the maximum execution time for the commit phase of this transaction.
- Deadlocks - the number of times this tranasaction deadlocked. Times for deadlocked transactions are not included in the latency, commit and prepare times.
- Rollback - the number of times this tranasaction rolled back. Times for rolled back transactions are not included in the latency, commit and prepare times.
- Exceptions - the number of times this tranasaction failed due to an unhandled exception. Times for transactions ended by exception are not included in the latency, commit and prepare times.
- Name - Either the name of the transaction, if provided to the Transaction constructor, or the name of the class, if the empty constructor was used.

The following snippet, which demonstrates both a named, and an unnamed transaction, was used to generate the system load:

```
// $Revision: 1.1.2.1 $
package com.kabira.snippets.tuning;

import com.kabira.platform.Transaction;

/**
 * SimpleTransactions example from the ActiveSpaces Transactions Tuning Guide
 * for generating transaction statistics.
 *
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */

public class SimpleTransactions
{
    public static void main(String[] args)
    {
        while (true)
```

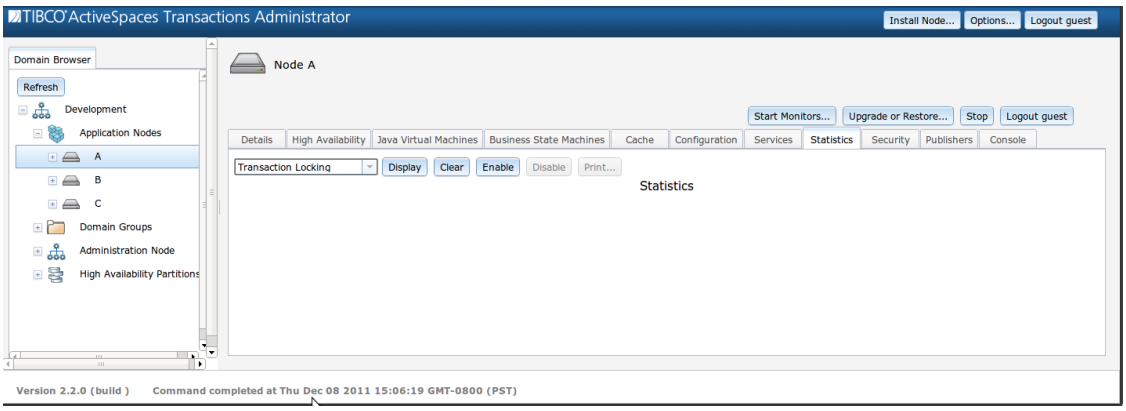
```
        {
            // This is a named transaction
            new Transaction("Example Transaction")
            {
                @Override
                public void run()
                {
                    try
                    {
                        Thread.sleep(500);
                    }
                    catch (InterruptedException ex) { }
                }
            }.execute();

            // This is an unnamed transaction
            new Transaction()
            {
                @Override
                public void run()
                {
                    try
                    {
                        Thread.sleep(500);
                    }
                    catch (InterruptedException ex) { }
                }
            }.execute();
        }
    }
}
```

The currently collected statistics may be cleared (reset) at any time by clicking the `Clear` button.

# Transaction locking and contention

A report showing transaction locking is available by selecting Transaction Locking in the pull down menu in the statistics panel:




## Figure 5.18. Transaction locking statistics

The collection of transaction locking statistics imposes a slight performance penalty and is not enabled by default. After having selected the Transaction Locking, enable the collection of these statistics by clicking the `Enable` button:


## Figure 5.19. Enable transaction locking statistics collection

Allow the data to collect for several seconds or more, and then disable statistics collection by clicking the `Disable` button. Disabling the statistics collection removes the slight performance penalty and allows the system to run at full speed.

Disabling the statistics collection does not remove the collected statistics.

Display the collected statistics by clicking the `Display` button:

```
======== Transaction Locking report for A.breakpoint ========

Data gathered between 2017-01-26 12:20:11 PST and 2017-01-26 12:20:52 PST.

                    Locks                Deadlocks          Acquire Time (usecs)
        Read    Write Promote  Cntion  Normal Promote Destroyed     Avg     Min     Max
Type Name
           0        3       0       0       0       0         1  5573.2     3.4 16712.2
com.x.MyManaged
           2       14       0       0       0       0         0     3.0     1.3     8.5
com.x.Factory
```

The columns of each row are:

- Type Name - the class for the Managed object.
- Read Locks - the number of transaction read locks that were taken on instances of this class during the period of statistics collection.
- Write Locks - the number of transaction write locks that were taken on instances of this class during the period of statistics collection.
- Promotions - the number of read locks that were promoted to write locks on instances of this class during the period of statistics collection. This is a subset of the number of write locks.
- Contentions - the number of transaction locks on this object that encountered transaction lock contention during the period of statistics collection.
- Deadlocks - the number of deadlocks other than promotion deadlocks that occurred attempting to transaction lock this object during the period of statistics collection.

- Promotion Deadlocks - the number of deadlocks that occurred attempting transaction lock promotion (from read to write) on this object during the period of statistics collection.
- Object Destroyed - the number of times a transaction lock failed due to the object having been destroyed in another transaction.
- Average Microseconds - the average time in microseconds that it took to acquire a transaction lock on this object during the period of statistics collection.
- Minimum Microseconds - the minimum time in microseconds that it took to acquire a transaction lock on this object during the period of statistics collection.
- Maximum Microseconds - the maximum time in microseconds that it took to acquire a transaction lock on this object during the period of statistics collection.

Transaction locking statistics should only be collected by one user at a time. Multiple concurrent collection will cause invalid data to be reported.

# Business State Machine

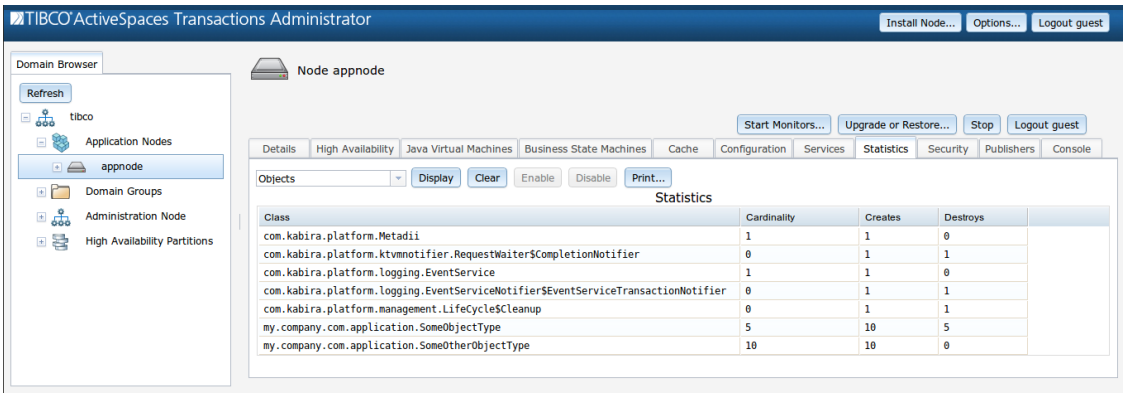
Documentation for the Business State Machine report can be found in the administration section of the site documentation for the Business State Machine component:

[http://downloads.fluency.kabira.com/sites/businessstatemachine/administration]

# Object Report

A report showing Managed objects statistics.

These statistics are available while the node is running.




### Figure 5.20. Managed object report

There is a row in the report for each Managed object type that has any non-0 data during since the data was last cleared. The columns in the report are:

- Class - the class name.
- Cardinality - the number of instances currently in shared memory.
- Creates - the number of instances created since the data was last cleared.

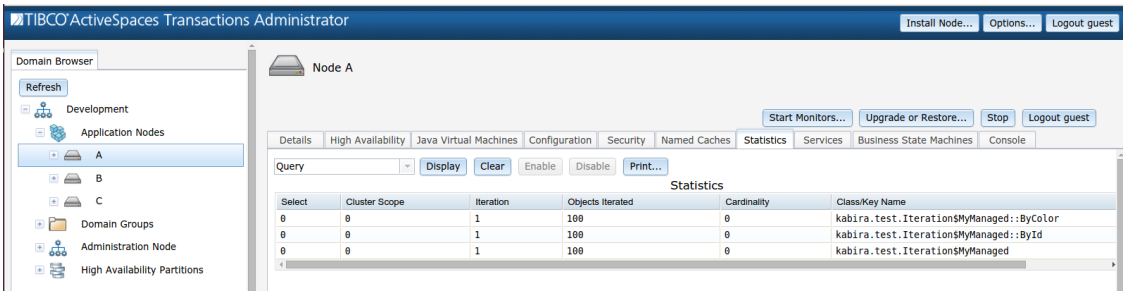
- Destroys - the number of instances destroyed since the data was last cleared.

The Creates and Destroys counts are non-transactional. They represent the number of times that these operations occurred without respect to the outcome of their containing transactions. For example, if one create was done, but a deadlock occurred, and the transaction rolled back, and then replayed successfully, the Creates count would be 2.

# Query Report

A report showing query counts for Managed objects.

These statistics are available while the node is running.




### Figure 5.21. Query statistics

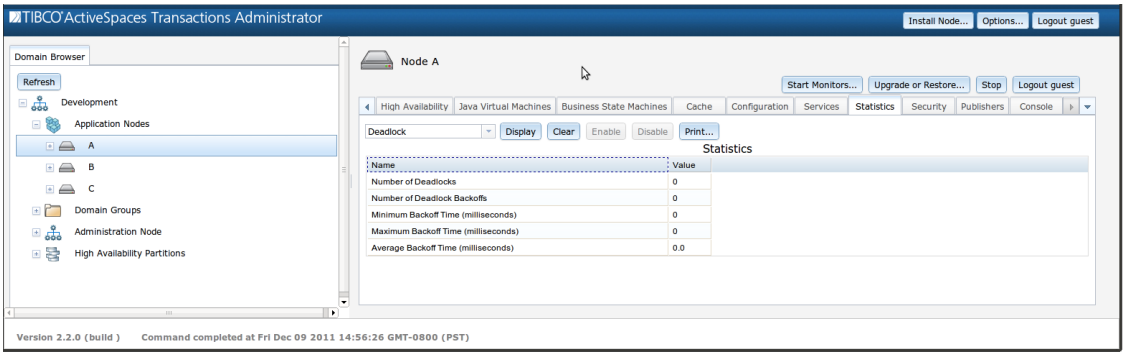
There is a row in the report for each Managed object type that has been accessed via `ManagedObject.extent()` or `ManagedObject.cardinality()`, and for each key type that has been accessed since the data was last cleared. The columns in the report are:

- Select - the number of times the Key was accessed during the period measured.
- Cluster Scope - the number of distributed queries during the period measured.
- Iteration - the number of times `ManagedObject.extent()` was called for the type during the period measured.
- Objects Iterated - the total number of objects iterated on results from `ManagedObject.extent()` or `KeyQuery.getResults()` during the period measured.
- Cardinality - the number of times `ManagedObject.cardinality()` was call for the type during the period measured.
- Class/Key Name - the type name (Extent) or the Key name.

# Deadlocks

When TIBCO ActiveSpaces® Transactions detects a deadlock, detailed information is sent to the log files. See the Analyzing Deadlocks section of the Chapter 4 for more information on interpreting this log data.

In addition to descriptive logging of each deadlock encountered, there are several statistics maintained that can been seen from selecting the Deadlock statistic in the statistics pull down menu, and clicking the `Display` button:




- Number of Deadlocks - the number deadlocks seen since the last time the statistics were cleared.
- Number of Deadlock Backoffs - the number of times an extra wait was inserted before a deadlock retry. For a given execution of a transaction, the first deadlock will be immediately aborted and retried. The next deadlock will incur a 100 millisecond wait time before it is retried. For each subsequent deadlock (in the same execution of a single transaction) the wait time will be doubled.
- Minimum Backoff Time (milliseconds) - the minimum amount of time, in milliseconds, a transaction spent waiting before retrying.
- Maximum Backoff Time (milliseconds) - the maximum amount of time, in milliseconds, a transaction spent waiting before retrying.
- Average Backoff Time (milliseconds) - the average amount of time, in milliseconds, a single transaction spent waiting before retrying.

These statistics are always collected, and may be cleared at any time by clicking the `Clear` button.

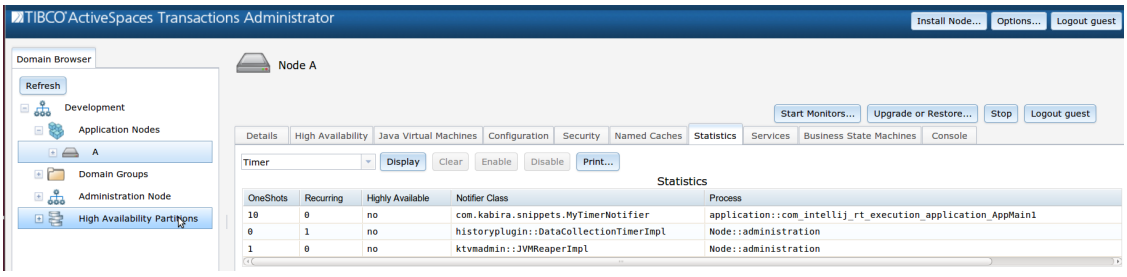
These statistics are available while the node is running.

All deadlocks are inherently performance hostile and should be avoided by modifying the application to avoid the deadlock condition.

# Timers

A report showing the timers currently queued in the node.

These statistics are available while the node is running.




There is a row in the report for each `com.kabira.platform.swtimer.TimerNotifier` class that has any timers currently started. The columns in the report are:

- OneShots - the number of one shot timers that are currently started with this notifier class.

- Recurring - the number of recurring timers that are currently started with this notifier class.
- Highly Available - whether or not the timer notifier class is highly available.
- Notifier Class - the class extending `com.kabira.platform.swtimer.TimerNotifier` used for the one shot and recurring timers.
- Process - the process (typically a JVM) where the timers were started.

# Node monitoring

A variety of reports are available within the statistics panel which apply to an entire node.

## Shared memory usage

Show the current shared memory utilization within the node.

These statistics are available while the node is running.

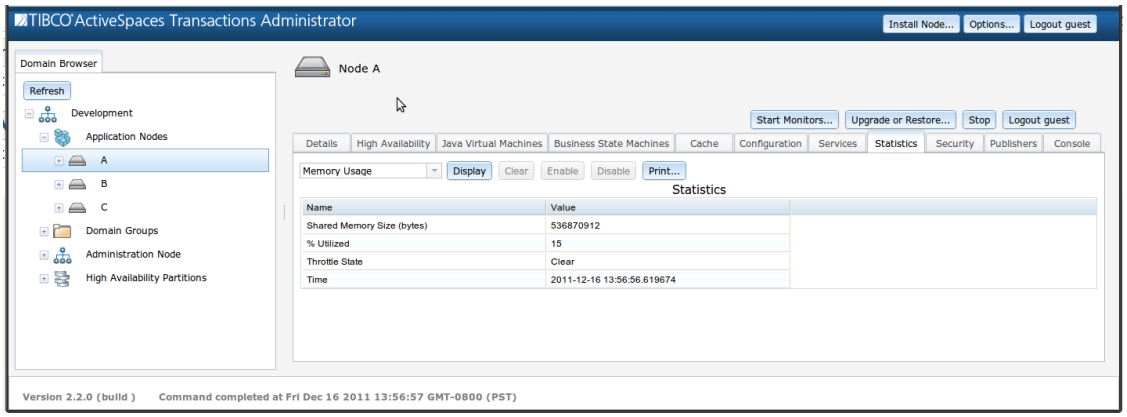



**Figure 5.22. Shared memory usage**

- Shared Memory Size - The total size of the shared memory, in bytes.
- % Utilized - Current percentage of shared memory that is in use.
- Throttle State - Historical list, in reverse order of memory throttling state changes. See `com.kabira.platform.swbuiltin.EngineServices.throttle()` for details about memory throttling.
- Time - The time of the memory throttling state change, or of the last check of the memory throttling state.

## Named caches

Show statistics for currently defined named caches within the node.

These statistics are available while the node is running.

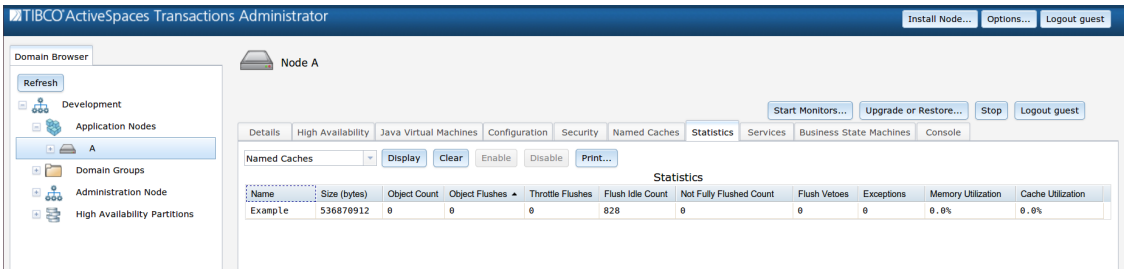



**Figure 5.23. Named caches**

- Name - The name of the cache.
- Size - The configured size of the cache in bytes.
- Object Count - The number of objects currently in the cache.
- Object Flushes - The number of objects that have been flushed since the last time the named cache statistics were cleared.
- Throttle Flushes - The number of objects that were flushed due to memory throttling since the last time the named cache statistics were cleared.
- Flush Idle Count- The number of times the flusher thread has run and found nothing to flush since the last time the named cache statistics were cleared.
- Not Fully Flushed Count - The number of times the flusher thread has run and not been able to bring the cache back to under the configured size since the last time the named cache statistics were cleared.
- Flush Vetoes - The number of object flushes that were vetoed by a FlushNotifier since the last time the named cache statistics were cleared.
- Exceptions - The number of exceptions that have occurred in FlushNotifiers since the last time the named cache statistics were cleared.
- Memory Utilization - What percentage of the entire shared memory is being used by the cache.
- Cache Utilization - What percentage of the configured cache size is currently being used.

# Shared memory hashes

Show current characteristics of system shared memory data structure hashes. The displaying of hash statistics, particulary if the hashes contain many buckets, may cause pauses in the system while the report is running.

These statistics are available while the node is running.

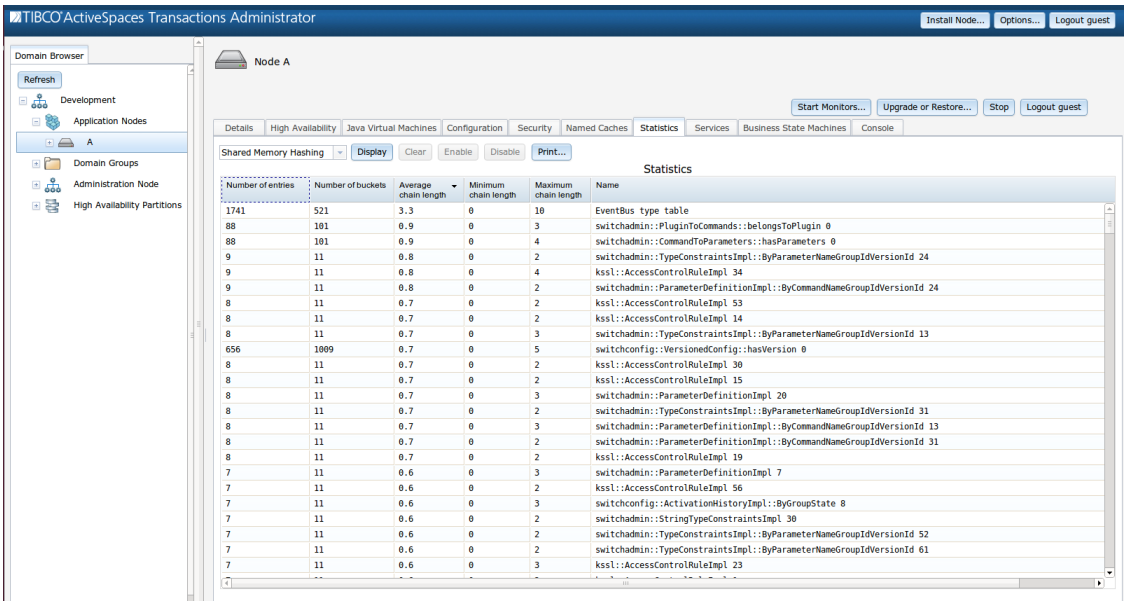



**Figure 5.24. Shared memory hashes**

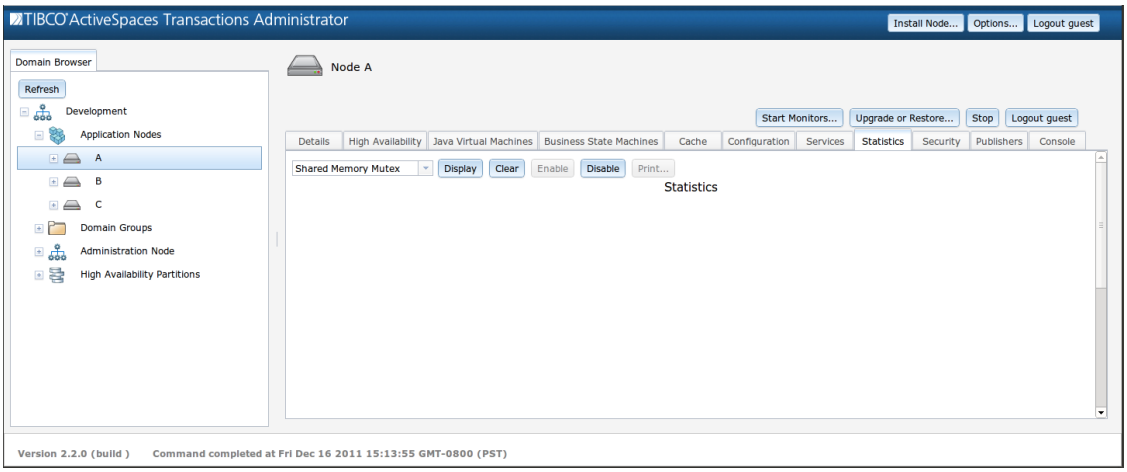
- Number of entries - The total number of items in the hash.
- Number of buckets - The number of hash buckets for the hash.
- Average chain length - The average length of the lists chained from each hash bucket.
- Minimum chain length - The minimum length for a list chained from any of the hash buckets.
- Maximum chain length - The minimum length for a list chained from any of the hash buckets.
- Name - The name of the system data structure.

# Shared memory mutex

A report showing shared memory system mutex locking is available by selecting Shared Memory Mutex in the pull down menu in the statistics panel, and enabling collection by clicking the `Enable` button. The collection of mutex locking statistics imposes a slight performance penalty and is not enabled by default.

These statistics are available while the node is running.

Note that mutex statistics report on low level runtime synchronization primitives that are not directly exposed to or manipulated by application code.

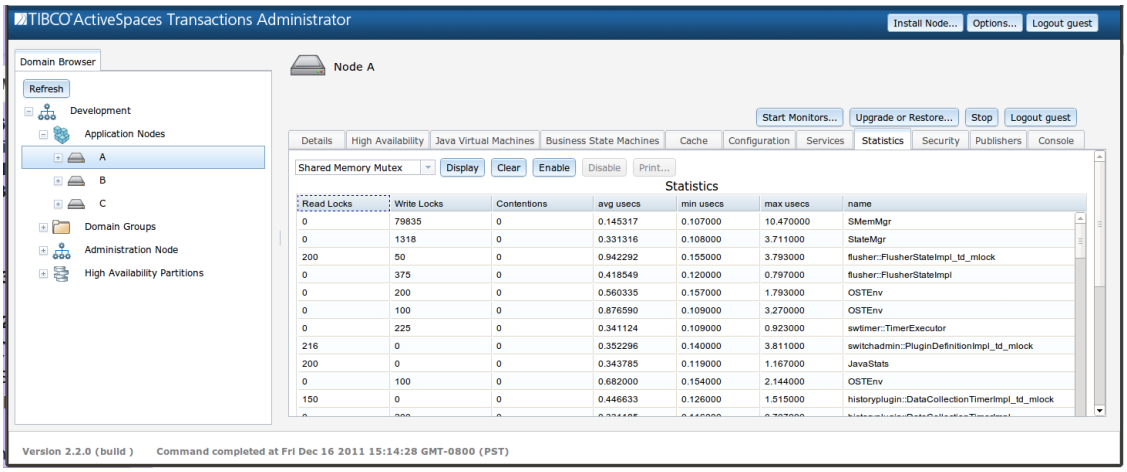



## Figure 5.25. Shared memory mutex enable

Allow the data to collect for several seconds or more, and then disable statistics collection by clicking the `Disable` button. Disabling the statistics collection removes the slight performance penalty and allows the system to run at full speed.

Disabling the statistics collection does not remove the collected statistics.

Display the collected statistics by clicking the `Display` button:




## Figure 5.26. Shared memory mutex display

- Read Locks - The number of times the lock was locked for shared read access since the last clear of the statistic.
- Write Locks - The number of times the lock was exclusively locked since the last clear of the statistic.
- Contentions - The number of times that acquiring either a read or a write lock encountered contention since the last clear of the statistic.
- avg usecs - The average number of microseconds taken to acquire the lock since the last clear of the statistic.

- min usecs - The minimum number of microseconds taken to acquire the lock since the last clear of the statistic.
- max usecs - The maximum number of microseconds taken to acquire the lock since the last clear of the statistic.
- name - A label associating a particular mutex with its owner or function in the system.

# Local mutex

A report showing process local mutex locking is available by selecting Local Mutex in the pull down menu in the statistics panel, and enabling collection by clicking the `Enable` button. The collection of mutex locking statistics imposes a slight performance penalty and is not enabled by default.

These statistics are available for running JVMs and system processes.

Note that local mutex statistics report on low level runtime synchronization primitives that are not directly exposed to or manipulated by application code.

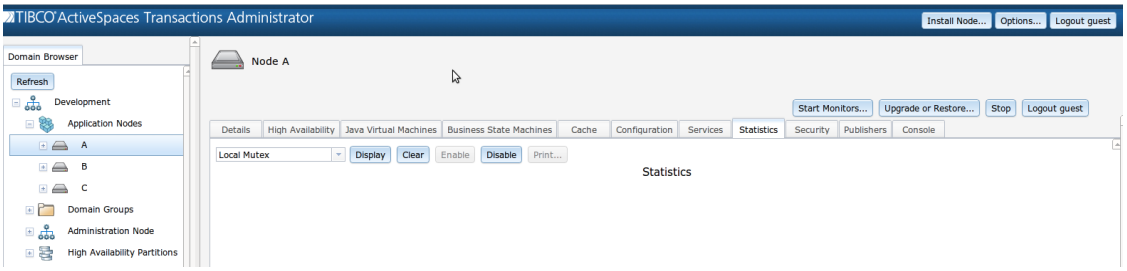



**Figure 5.27. Process local mutex enable**

Allow the data to collect for several seconds or more, and then disable statistics collection by clicking the `Disable` button. Disabling the statistics collection removes the slight performance penalty and allows the system to run at full speed.

Disabling the statistics collection does not remove the collected statistics.

Display the collected statistics by clicking the `Display` button:

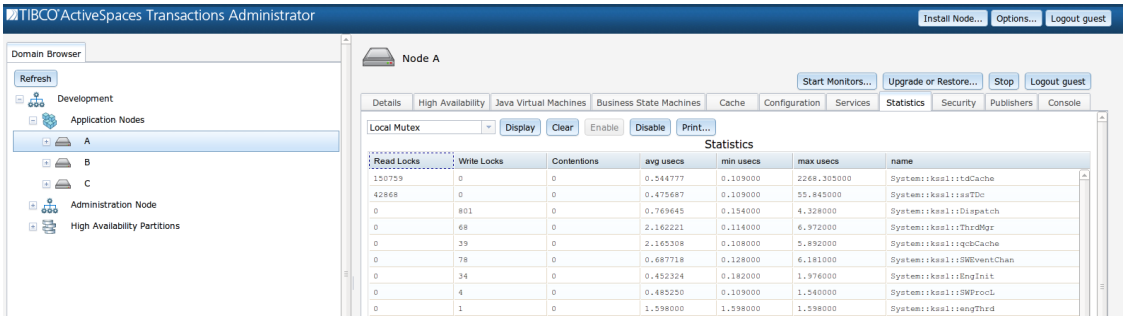



**Figure 5.28. Process local mutex display**

- Read Locks - The number of times the lock was locked for shared read access since the last clear of the statistic.

- Write Locks - The number of times the lock was exclusively locked since the last clear of the statistic.
- Contentions - The number of times that acquiring either a read or a write lock encountered contention since the last clear of the statistic.
- avg usecs - The average number of microseconds taken to acquire the lock since the last clear of the statistic.
- min usecs - The minimum number of microseconds taken to acquire the lock since the last clear of the statistic.
- max usecs - The maximum number of microseconds taken to acquire the lock since the last clear of the statistic.
- name - A label associating a particular mutex with its owner or function in the system. The value is prefixed with the name of the process or JVM which contains the mutex.

# Shared Memory IPC

The runtime uses a shared memory based interprocess communication mechanism for a within-node RPC mechanism, and also for managing the life cycle of asynchronous method calls.

These statistics are available while the node is running.

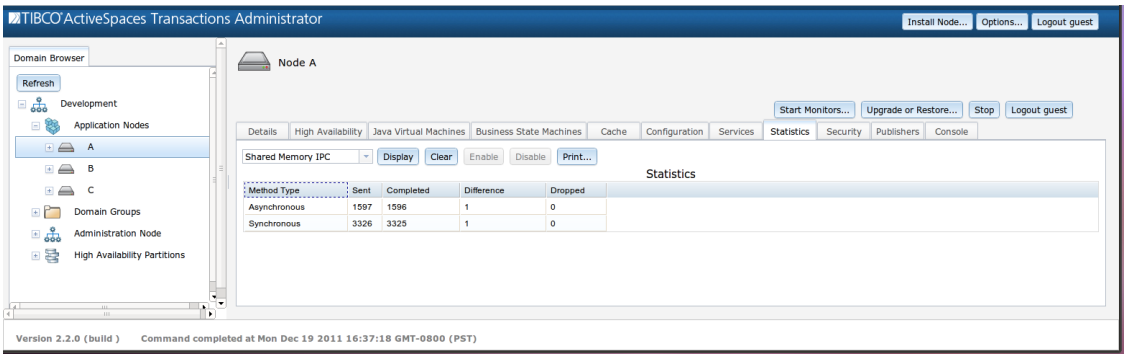



**Figure 5.29. Shared Memory IPC**

- Method Type - Synchronous or Asynchronous.
- Sent - The number of method invocations.
- Completed - The number of method invocations that have completed execution.
- Difference - The difference between the number of invocations, and the number of completed invocations. For asynchronous methods, this can show queuing.
- Dropped - The number of queued method invocations which were dropped because the target object had been destroyed.

# Shared Memory IPC Detailed

Finer grained shared memory IPC statistics, showing each of the methods invoked within a node are also available. The collection of these statistics imposes a slight performance penalty and consumes shared memory for each method invocation, and is not enabled by default.

These statistics are available while the node is running.

To collect these statistics, select Shared Memory IPC Detailed in the pull down menu in the statistics panel:

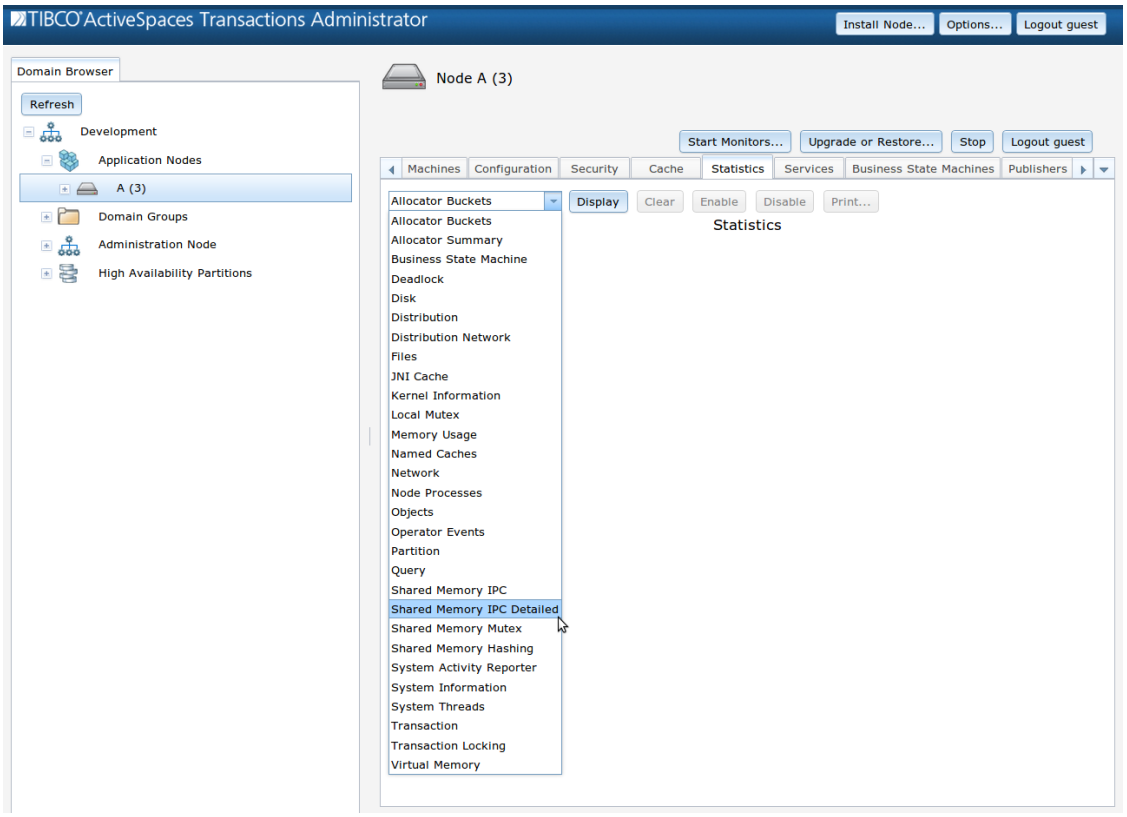



**Figure 5.30. Shared memory IPC detailed**

Next enable the collection of these statistics by clicking the `Enable` button. After you have collected statistics for a sufficient period of time press the `Disable` button to stop the statistics collection. Pressing the `Clear` button clears the currently collected statistics and reclaims the memory used by the statistics collection.

The statistics may be displayed at any time by pressing the `Display` button.

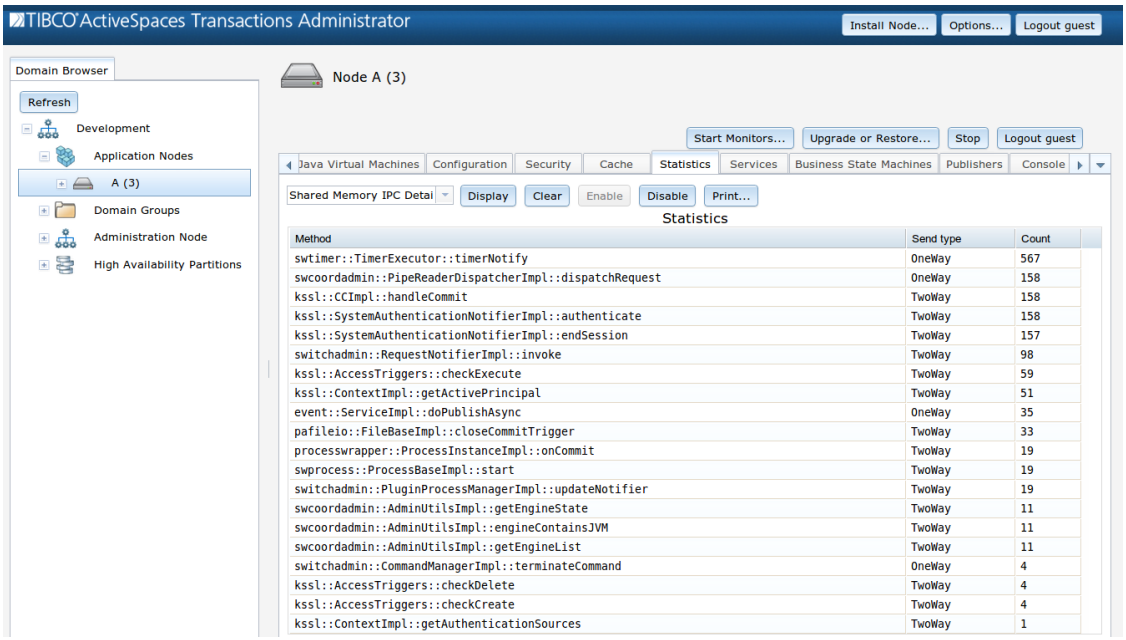



**Figure 5.31. Shared Memory IPC Detailed**

- Method - The name of the method invoked via the shared memory IPC mechanism.
- Send Type - Synchronous or Asynchronous.
- Count - The number of method invocations.

# Partition Report

Display information about the High Availability Partitions currently defined in the node.

These statistics are available if the node has been configurated for distribution and there is at least one JVM running.

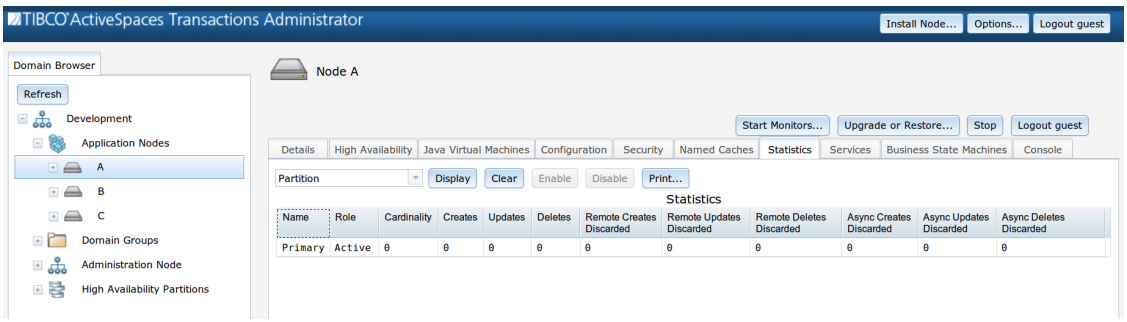



**Figure 5.32. High Availability Partitions Report**

- Name - The name of the partition.
- Role - Whether the partition active on this node, or is a replica.
- Cardinality - The number of objects currently in the partition.

- Creates - The number of times that an object was created in this partition since the last clear of the statistics. This statistic is not transactional.
- Updates - The number of times that an object was modified in this partition since the last clear of the statistics. This statistic is not transactional.
- Updates - The number of times that an object was deleted in this partition since the last clear of the statistics. This statistic is not transactional.
- Remote Creates Discarded - The number of creates since the last clear of the statistics, that could not be sent to a remote node.
- Remote Updates Discarded - The number of updates since the last clear of the statistics, that could not be sent to a remote node.
- Remote Deletes Discarded - The number of deletes since the last clear of the statistics, that could not be sent to a remote node.
- Async Creates Discarded - The number of asynchronous create failures that occurred in this partition since the last clear of the statistics.
- Async Updates Discarded - The number of asynchronous update failures that occurred in this partition since the last clear of the statistics.
- Async Deletes Discarded - The number of asynchronous delete failures that occurred in this partition since the last clear of the statistics.

# Shared Memory System Allocations Report

A report of all system shared memory allocations can be found in the Allocations Summary.

These statistics are available while the node is running.

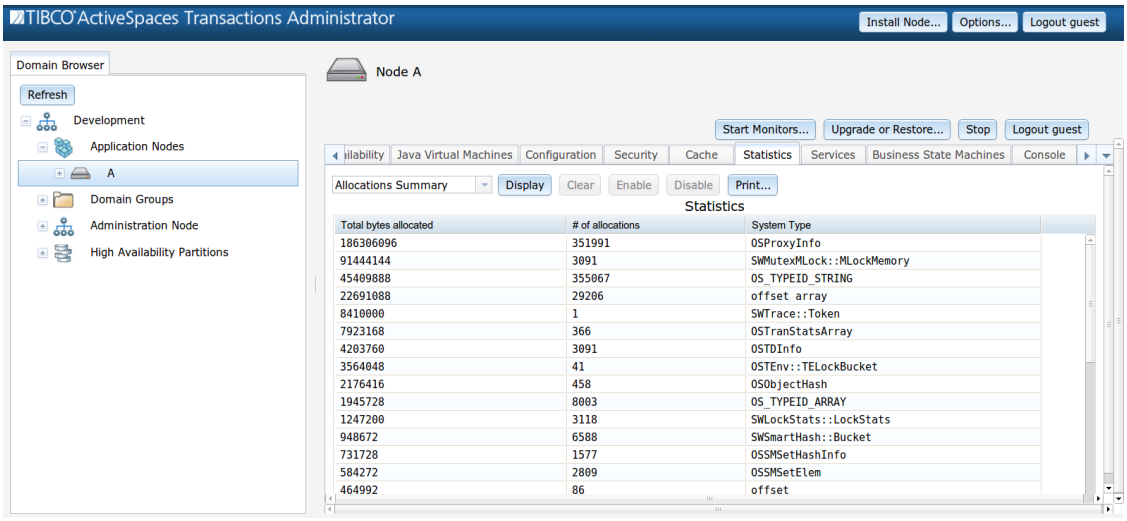



**Figure 5.33. Shared Memory Allocations Summary**

- Total bytes allocated - the total size, in bytes, of all current allocations of this system type.
- # of allocations - the total number of allocations of this system type.

- System Type - the system type allocated.

# Shared Memory Allocator Report

General details about the shared memory allocator can be found in the Allocator Summary report

These statistics are available while the node is running.

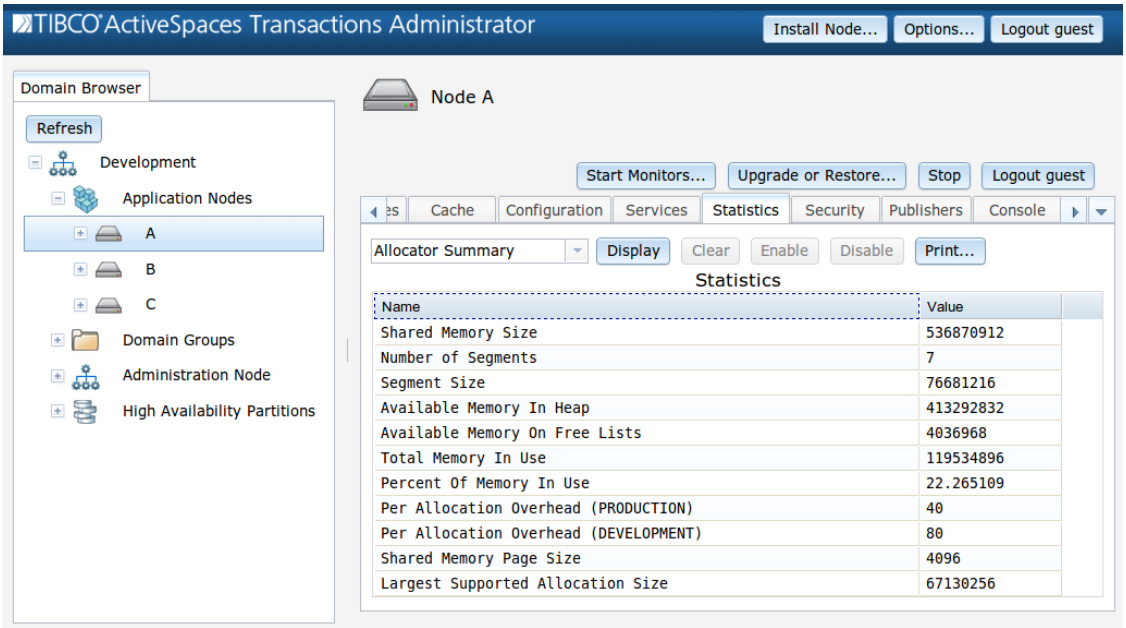



**Figure 5.34. Shared Memory Allocator Summary**

- Shared Memory Size - the size, in bytes, of the node's shared memory.
- Number of Segments - the number of segments (degree of parallelism) in the unallocated heap.
- Segment Size - the initial size, in bytes, of each segment within the unallocated heap.
- Available Memory In Heap - the amount, in bytes, of shared memory that has not yet been used by the node.

  See the following Shared Memory Allocator Buckets Report section for details about the organization of the shared memory allocator.
- Available Memory On Freelists - the amount, in bytes, of node shared memory that has already been allocated and deallocated, and is now available on the freelist.

  See the following Shared Memory Allocator Buckets Report section for details about the organization of the shared memory allocator.
- Total Memory In Use - the amount, in bytes, of shared memory that is currently in use on the node.
- Percent Of Memory In Use - Total Memory In Use shown as a percentage of Shared Memory Size.

- Per Allocation Overhead (PRODUCTION) - The amount, in bytes, of shared memory allocation bookkeeping per allocation, for a node running with with a PRODUCTION build.
- Per Allocation Overhead (DEVELOPMENT) - The amount, in bytes, of shared memory allocation bookkeeping per allocation, for a node running with with a DEVELOPMENT build.
- Shared Memory Page Size - Not currently used.
- Largest Supported Allocation Size - The largest individual shared memory allocation size, in bytes, supported on this node.

# Shared Memory Allocator Buckets Report

In a freshly started node, all shared memory starts in the shared memory heap. When shared memory is first allocated, it taken from the heap. When shared memory is freed, it is put on a freelist, organized allocation size. Subsequent allocations will first attempt to find memory on the freelist.

Detailed information about the current state of shared memory allocations and freelists may be found in the allocator buckets report

These statistics are available while the node is running.

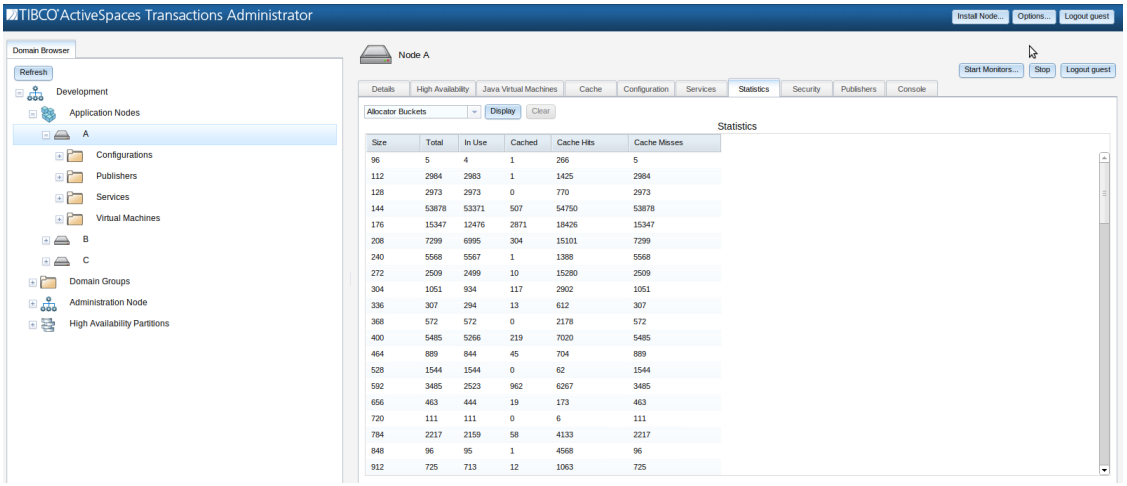



**Figure 5.35. Shared Memory Allocator Buckets Report**

- Size - the number of bytes for this allocation bucket.
- Total - the number of allocations of this size that have been taken from the shared memory heap.
- In Use - the number of allocations of this size currently in use.
- Cached - the number of allocations of this size that are currently on the freelist.
- Cache Hits - the number of times an allocation request for this size was made and filled from the freelist.
- Cache Misses - the number of times an allocation request for this size was filled from the shared memory heap.

# System Threads Report

Display information about system threads within the node.

These statistics are for all running JVMs and system processes.

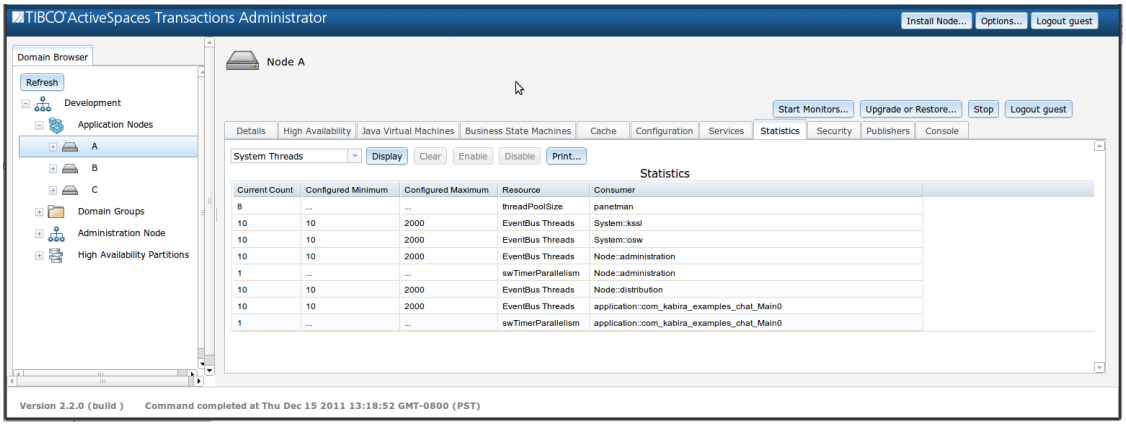



**Figure 5.36. System Threads Report**

- Current Count - The number of threads that currently exist.
- Configured Minimum - The configured value for the minimum number of this type of system thread.
- Configured Maximum - The configured value for the maximum number of this type of system thread.
- Resource - The system resource associated with this thread.
- Consumer - The process or system component associated with this thread resource.

# Files Report

Display the files in the node directory, and their sizes in bytes.

These statistics are available while the node is running.

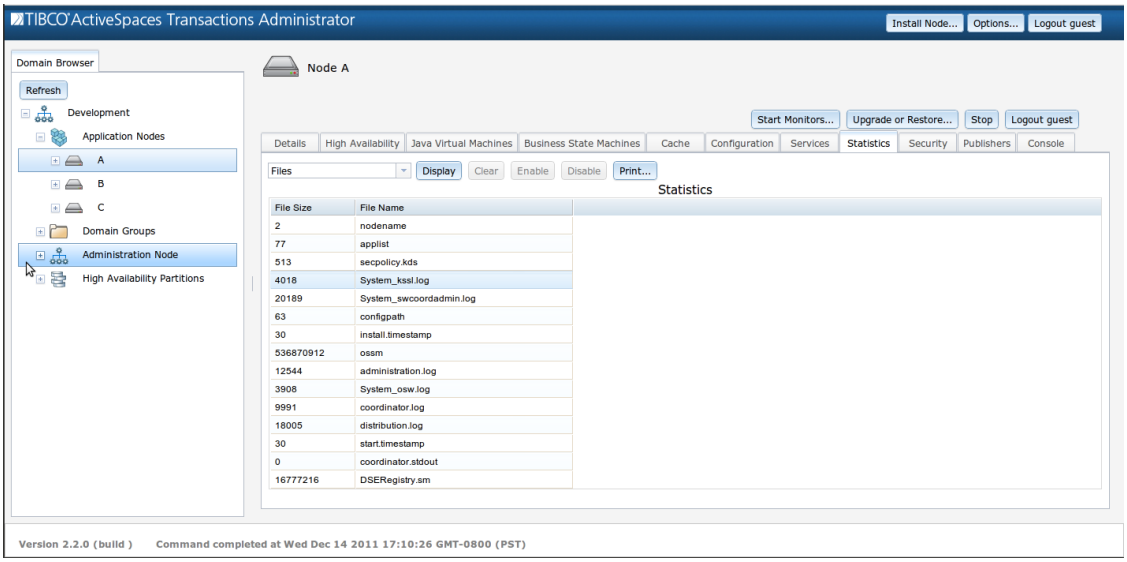



**Figure 5.37. Files Report**

# Node Processes Report

Display a list of the application, node, and system processes running on the node.

These statistics are available while the node is running.

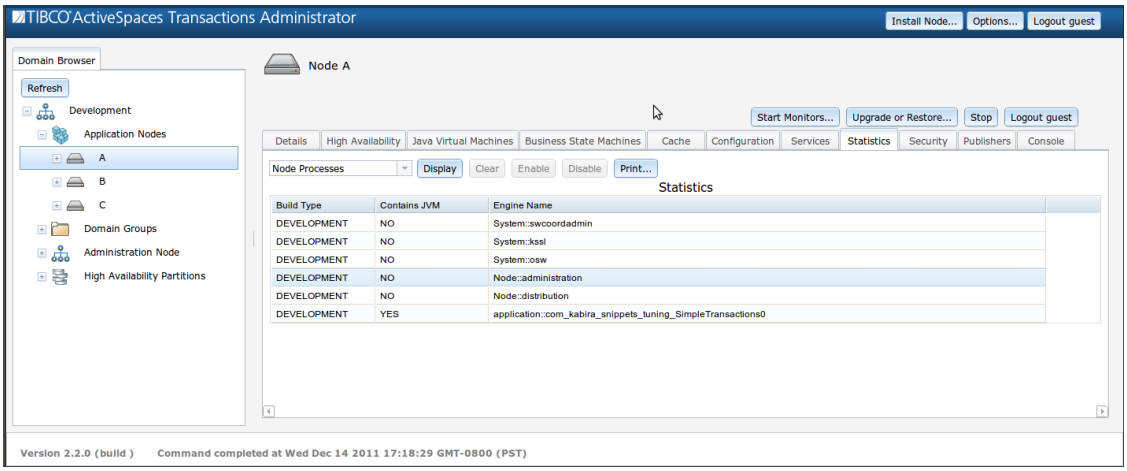


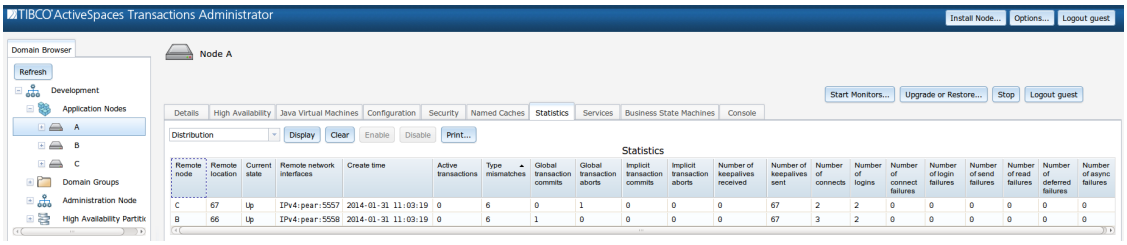

**Figure 5.38. Node Processes Report**

- Build Type - Whether the process contains PRODUCTION binaries or DEVELOPMENT (debug) binaries.
- Contains JVM - Whether or not this process contains a Java VM.
- Process Name - The name of the process.

# Distribution Report

State information about the TIBCO ActiveSpaces® Transactions distribution layer is available by selecting `Distribution` in the pull down menu in the statistics panel.

These statistics are available if the node has been configured for distribution and there is at least one JVM running.




## Figure 5.39. Distribution report

- Remote node - which other node in the cluster that this row of statistics refer to.
- Remote location - the location code for the remote node.
- Current state - this node's view of the current state of the remote node.
- Remote network interfaces - the network interface devices used for connections with the remote node.
- Create time - when the remote node was discovered by the current node.
- Active transactions - the number of transactions with the remote node currently in progress.
- Type mismatches - the number of types that are mismatched between the current node and the remote node.
- Global transaction commits - the total number of distributed transactions started and committed on the current node that involved the remote node. Clear resets this counter to 0.
- Global transaction aborts - the total number of distributed transactions started and aborted on the current node that involved the remote node. Clear resets this counter to 0.
- Implicit transaction commits - the total number of distributed transactions started and committed by the remote node that involved the current node. Clear resets this counter to 0.
- Implicit transaction aborts - the total number of distributed transactions started and aborted by then remote node that involved the current node. Clear resets this counter to 0.
- Number of keepalives received - the total number of internal keepalive messages received from the remote node. Clear resets this counter to 0.
- Number of keepalives sent - the total number of internal keepalive messages sent from this node to the remote node. Clear resets this counter to 0.
- Number of connects - the number of successful connections from this node to the remote node. Clear resets this counter to 0.

- Number of logins - the number of successful logins from the remote node to this node. Clear resets this counter to 0.
- Number of connect failures - the number of unsuccessful connections from this node to the remote node. Clear resets this counter to 0.
- Number of login failures - the number of unsuccessful logins from the remote node to this node. Clear resets this counter to 0.
- Number of send failures - the number of failed attempts to send data from this node to the remote node. Clear resets this counter to 0.
- Number of read failures - the number of failed attempts to read data from the remote node. Clear resets this counter to 0.
- Number of deferred failures - the number of failures attempting to write data to a failed remote node. Clear resets this counter to 0.
- Number of async failures - the number of failures attempting to replica asynchronous data to a failed remote node. Clear resets this counter to 0.

# Distribution Network Statistics

Network usage statistics by the TIBCO ActiveSpaces® Transactions distribution layer are available by selecting `Distribution Network` in the pull down menu in the statistics panel. The collection of network usage statistics imposes a slight performance penalty and is not enabled by default.

These statistics are available while the node is running.

To enable, click `Enable`.


**Figure 5.40. Distribution Network statistics enable**

Allow the data to collect for the desired amount of time and then disable the collection by clicking the `Disable` button.

Disabling network statistics collection does not clear the collecting data.

Display the collected statistics by clicking the `Disply` button.

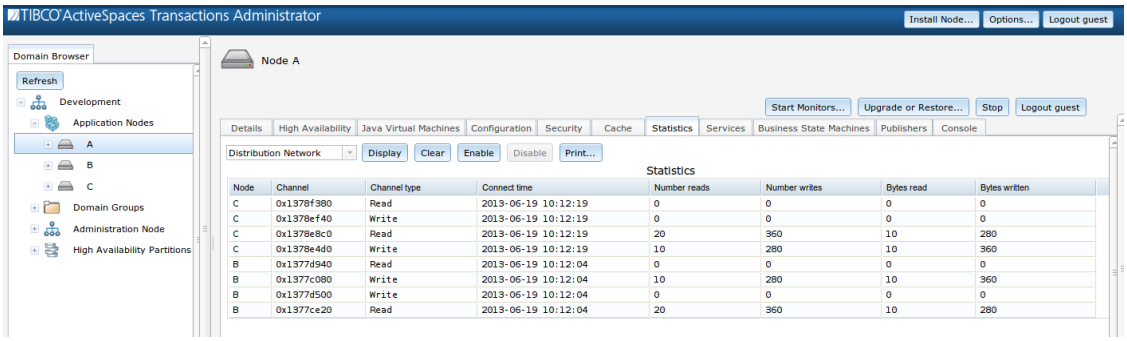



**Figure 5.41. Distribution Network statistics enable**

- Node - the name of the remote node that this channel (socket) is connected to.
- Channel - the name of the channel (socket).
- Channel type - the type of the channel (read or write).
- Connect time - when this channel's connection was established.
- Number reads - the total number of read calls on this channel. Clear resets this count to 0.
- Number writes - the total number of write calls on this channel. Clear resets this count to 0.
- Bytes read - the total number of bytes read from this channel. Clear resets this count to 0.
- Bytes written - the total number of bytes written to this channel. Clear resets this count to 0.

# JVM Native Runtime Calls

Show per-JVM information about native calls being made across JNI into the runtime.

These statistics are available for all running JVMS in the node.

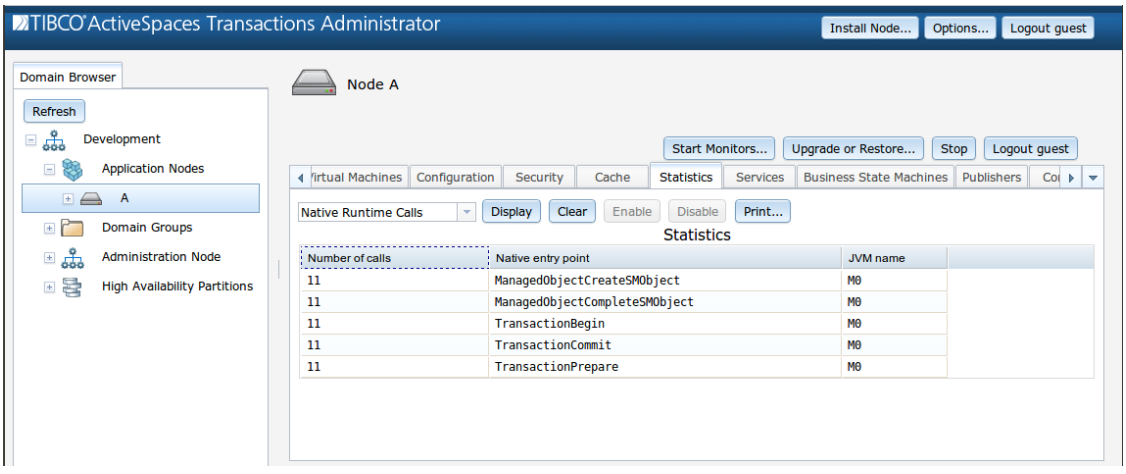



**Figure 5.42. Native runtime calls**

- Number of calls - Number of calls made since the last time this statistic was cleared.

- Native entry point - Name of the runtime native entry point.
- JVM name - Name of the JVM.

# Runtime JNI Calls

A report showing per-JVM information about JNI calls being made from the runtime into the JVM is available by selecting Runtime JNI Calls in the pull down menu in the statistics panel, and enabling collection by clicking the `Enable` button. The collection of runtime JNI statistics imposes a performance penalty and is not enabled by default.

These statistics are available for all running JVMS in the node.

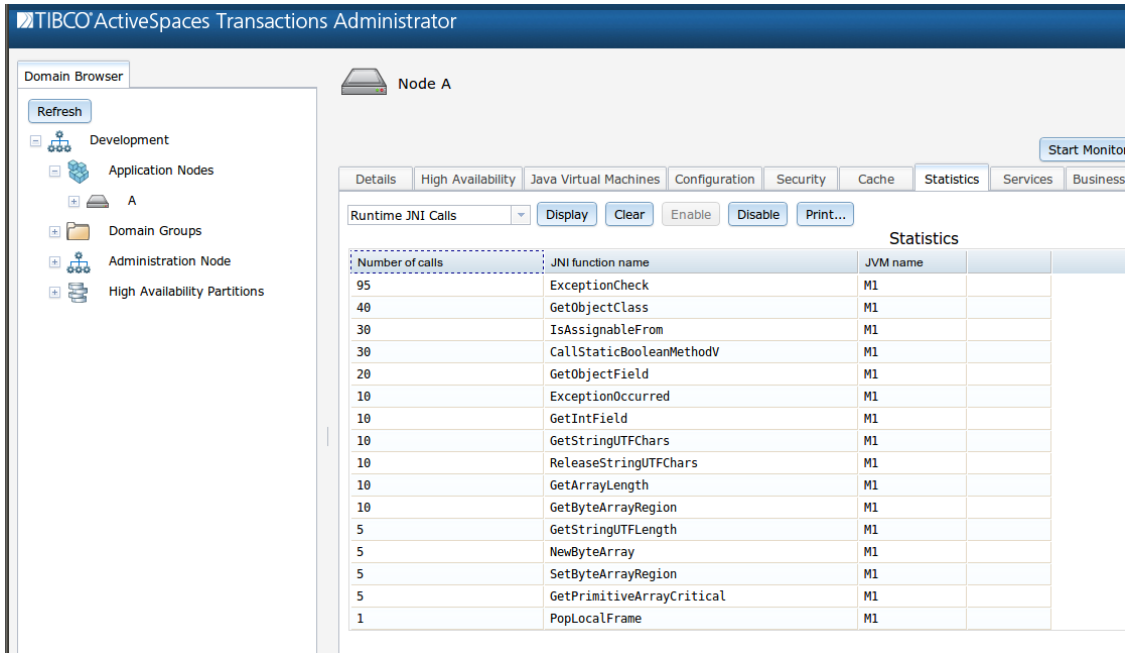



**Figure 5.43. Runtime JNI calls**

- Number of calls - Number of calls made since the last time this statistic was cleared.
- JNI function name - Name of the JNI function.
- JVM name - Name of the JVM.

# JNI Cache Statistics

Show information about the per-thread JNI caching done by the runtime for each JVM in the node.

These statistics are available for running JVMs and system processes.


**Figure 5.44. JNI Cache statistics**

- Thread Type - Shows whether the thread is a Java thread or a runtime thread.
- % Current - Current number of threads that have cached JNI resources.
- % Allocations - Number of JNI cache resource allocations that have been done since the last time this statistic was cleared.
- % Frees - Number of JNI cache resource deallocations that have been done since the last time this statistic was cleared.
- % JVM - The JVM that the resources are being cached for.

# System monitoring

## Kernel information

Show information about the version of the operating system.

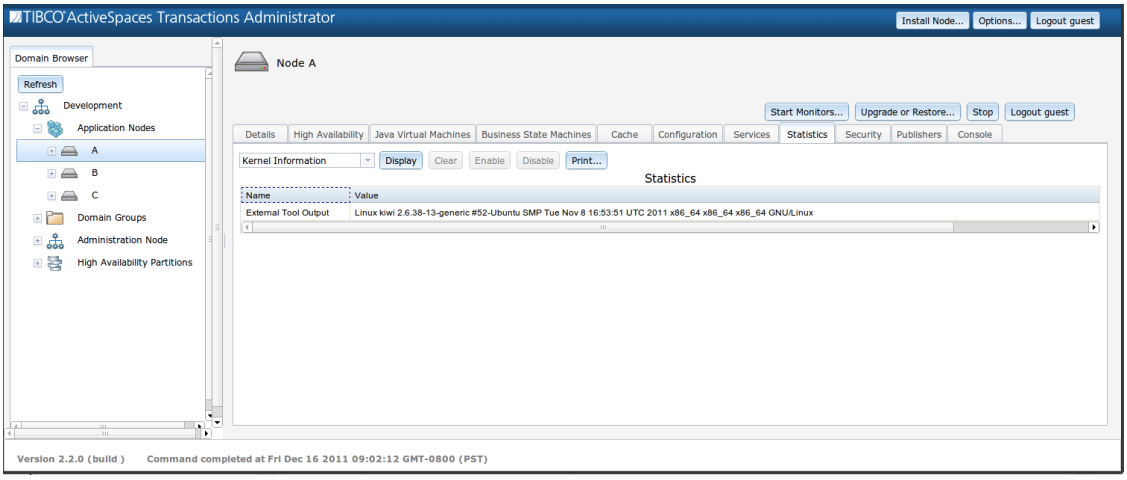



**Figure 5.45. Kernel information**

## System information report

Show information about the system, including the number of CPUs and their speed, the amount of physical memory, and shared memory mapping.

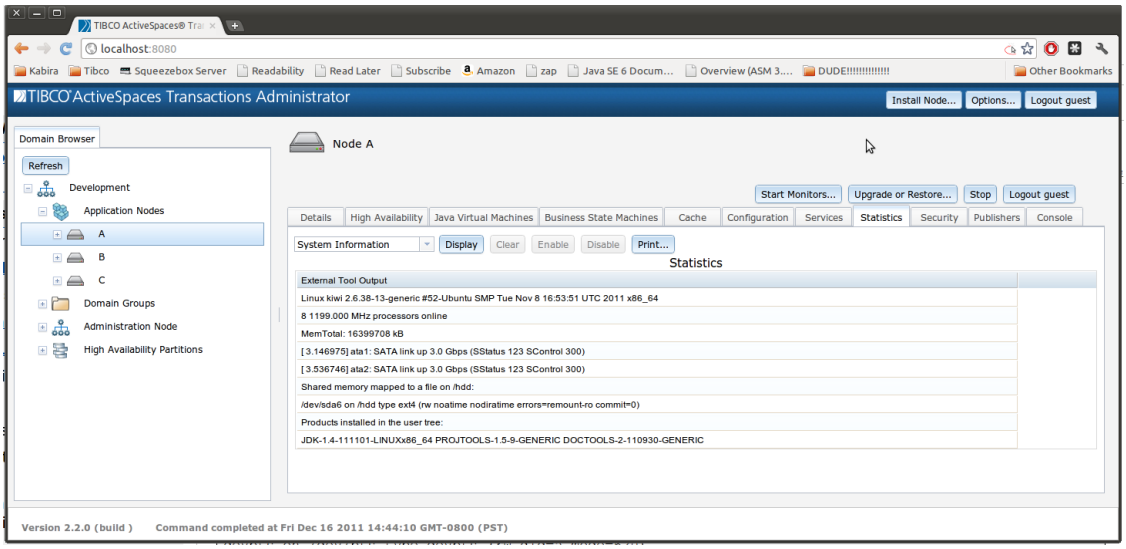



**Figure 5.46. System information**

# Virtual Memory Report

The Virtual Memory Report directly captures the output of a platform specific tool. On Unix systems this is `vmstat` run with a 1 second sampling interval.

Because running this tool consumes a small amount of CPU and an amount of disk space that is proportional to how long the tool is run, it is not enabled by default. To enable, select `Virtual Memory` in the pull down menu in the statistics panel and click the `Enable` button:

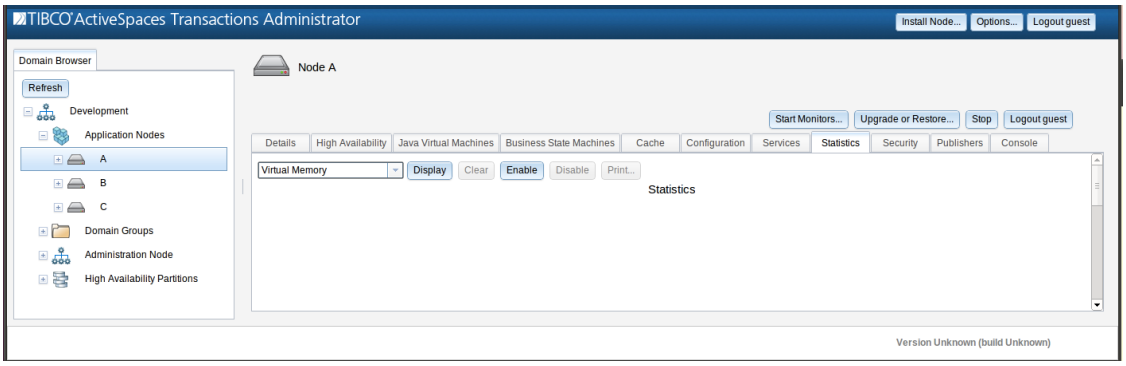



**Figure 5.47. Virtual Memory Report Enable**

Allow the data to collect for the desired amount of time and then disable the collection by clicking the `Disable` button.

Display the collected statistics by clicking the `Display` button.

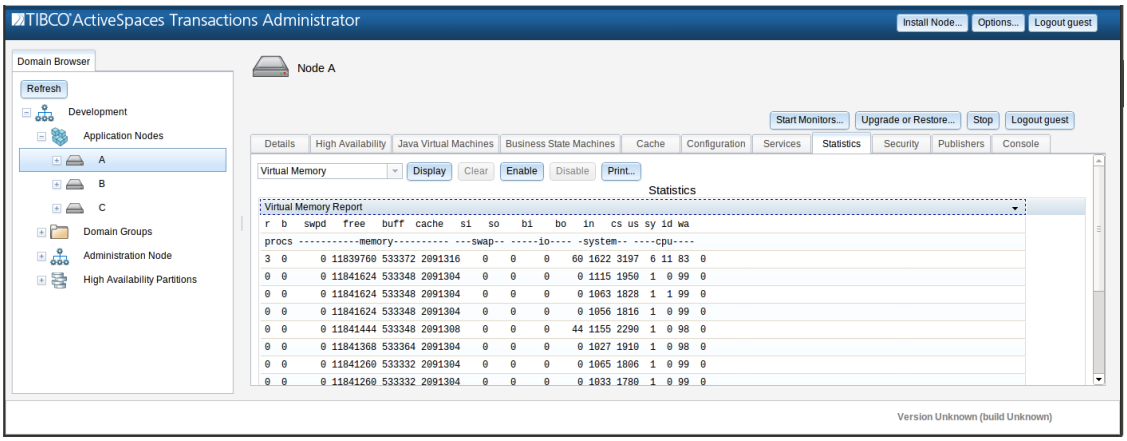



**Figure 5.48. Virtual Memory Report**

# Network Utilization Report

The Network Utilization Report directly captures the output of a platform specific tool. On Unix systems this is `nicstat`. run with a 2 second sampling interval.

Because running this tool consumes a small amount of CPU and an amount of disk space that is proportional to how long the tool is run, it is not enabled by default. To enable, select `Network` in the pull down menu in the statistics panel and click the `Enable` button:

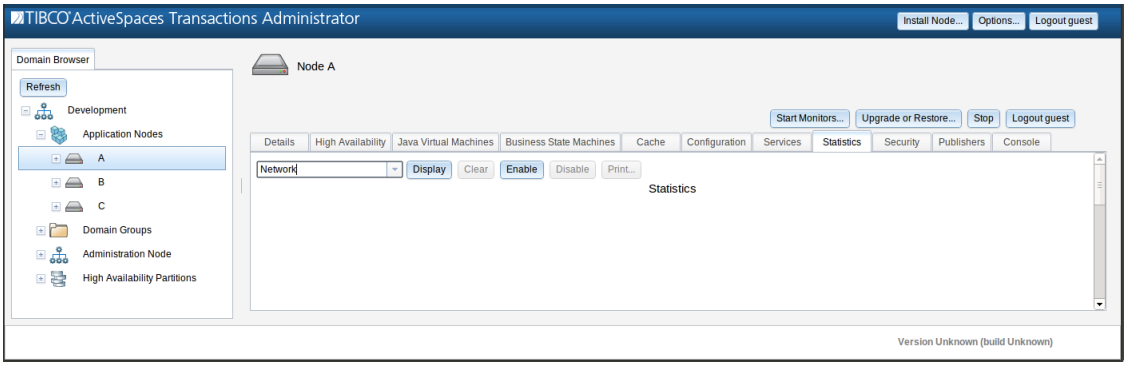



**Figure 5.49. Network Report Enable**

Allow the data to collect for the desired amount of time and then disable the collection by clicking the `Disable` button.

Display the collected statistics by clicking the `Display` button:

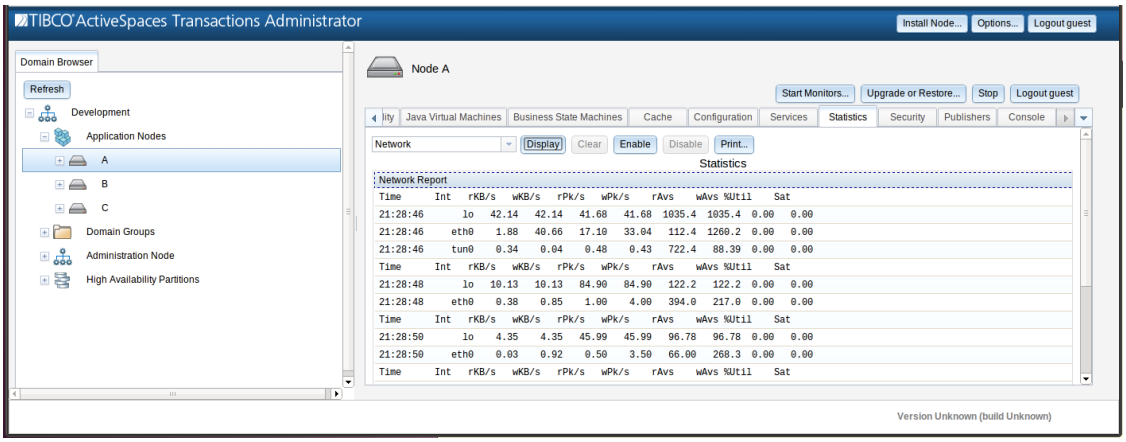



**Figure 5.50. Network Utilization Report**

# Disk Utilization Report

The disk utilization report directly captures the output of a platform specific tool. On Unix systems this is `iostat` run with a 2 second sampling interval.

Because running this tool consumes a small amount of CPU and an amount of disk space that is proportional to how long the tool is run, it is not enabled by default. To enable, select `Disk` in the pull down menu in the statistics panel and click the `Enable` button:

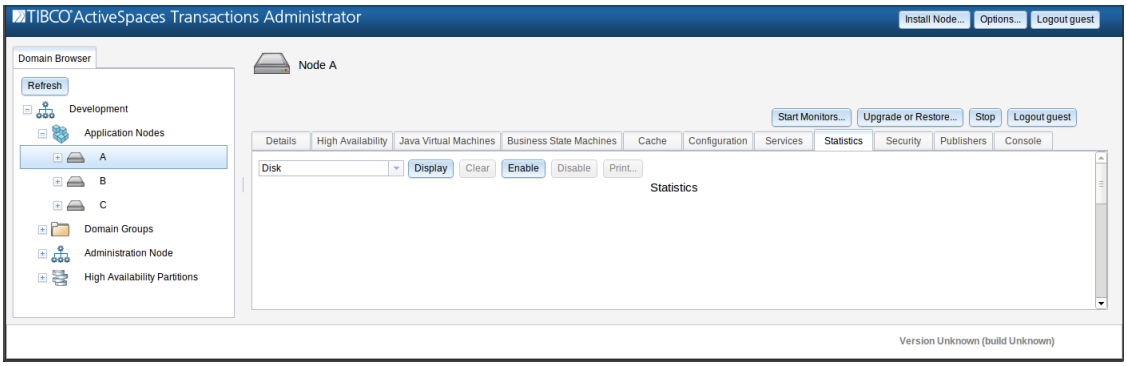



**Figure 5.51. Disk Report Enable**

Allow the data to collect for the desired amount of time and then disable the collection by clicking the `Disable` button.

Display the collected statistics by clicking the `Display` button:

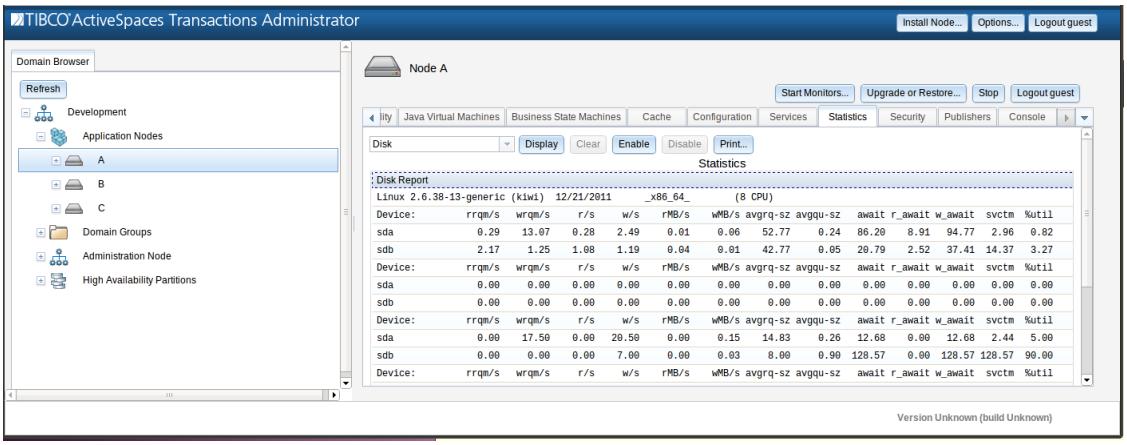



**Figure 5.52. Disk Report**

# System Activity Report

The System Activity Report directly captures the output of a platform specific tool. On Unix systems this is `sar`.

Because running this tool consumes a small amount of CPU and an amount of disk space that is proportional to how long the tool is run, it is not enabled by default. To enable, select `System Activity Reporter` in the pull down menu in the statistics panel and click the `Enable` button:

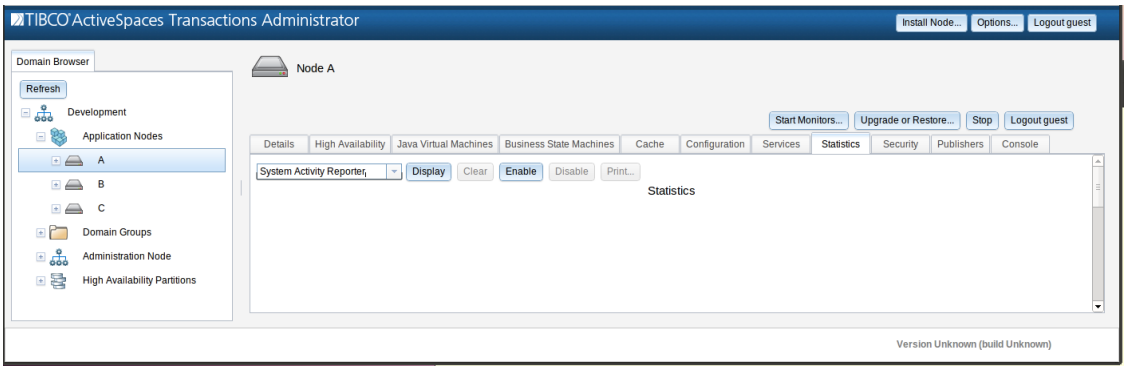



**Figure 5.53. System Activity Report Enable**

Allow the data to collect for the desired amount of time and then disable the collection by clicking the `Disable` button.

Display the collected statistics by clicking the `Display` button:

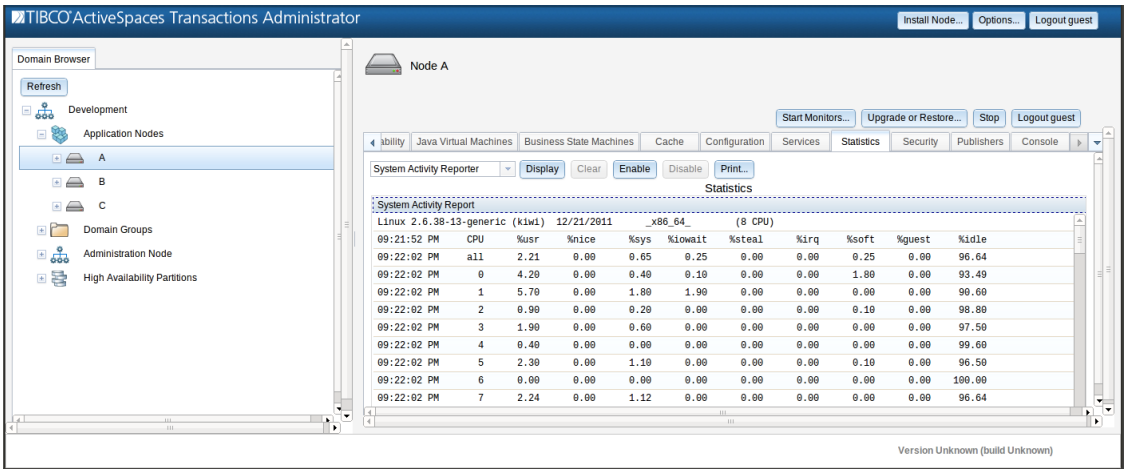



**Figure 5.54. System Activity Report**

# System impact of monitoring

## Statistics collection impact

If you look through the chapter of the Tuning Guide, where the individual statistic reports are described, you will see some of them containing sentences like:

```
The collection of mutex locking statistics imposes a slight performance
penalty and is not enabled by default.
```

Some statistics collection, turns on extra code/storage paths in the runtime. For each of these, an effort has been made in the runtime to minimize the costs, but they are clearly non-0. Application performance measurement is the best way to characterize these effects.

For a few of the statistics (e.g. eventbus detailed=true), there is also unbounded memory consumption. In the documentation you will see a statement like:

```
The collection of these statistics imposes a slight performance penalty
and consumes shared memory for each method invocation, and is not enabled
by default.
```

This is an area where, by leaving the statistic enabled, one risks running the node out of memory.

## Statistics reporting impact

Generally, statistics reporting can be more expensive (both computationally and in terms of contention) than statistics collection. There reasons for this are:

- Most statistics are stored as a set of integer values. Relatively inexpensive to update. But nearly all reporting involves row by row string formatting of the statistic data, often including type lookups.
- The synchronization for the collection of some statistics (particularly those in performance sensitive paths) is parallelized to allow concurrent writers. Where possible, this is done using atomic memory counter primitives, otherwise using pools of mutexes, or in some cases a single

mutex. For the runtime execution path, the locking is minimized. But the for statistics clear path, in paths where the statistic is protected by mutexes, one or all of the mutexes get locked. The data collection with the worst impact on the system would be the allocator report with the detailed=true option. This data reporting can cause expensive contention in the shared memory allocator.

- The returning of report data through the administration plugin service uses the same runtime support that an application does. Creating objects, consuming shared memory, etc... A large report (generally those that use the detailed=true option) can temporarily consume a great deal of shared memory.

# Recommendations

Run statistics reporting judiciously in a production system. Do not design an operator console where a room full of operators are all continuously issuing statistics commands.

Unless there is a good reason, avoid the detailed=true reporting.

**Measure**. Using your existing application performance testing, measure the impact of collecting the desired statistics. Also measure the impact of reporting the desired statistics.

# Index

## U