# Oracle J2EE State Replication Guidelines and Best Practices

*An Oracle White Paper*
*October 2007*

Maximum Availability Architecture

Oracle Best Practices
For High Availability

# Oracle J2EE State Replication Guidelines and Best Practices

# Oracle J2EE State Replication Guidelines and Best Practices

## EXECUTIVE SUMMARY

Oracle's J2EE State Replication provides reliability for HTTP Sessions and their associated objects. Application state held in memory by one  instance of the application can be configured to automatically be replicated to another instance of the application. Should the first instance become unavailable, the replication framework provides for transparent failover to another instance of the application with the replicated session state.

The advantages of State Replication include application availability, transparent failover and the load balancing provided by Oracle's J2EE Cluster framework. Oracle Application Server 10g R3 provides a standards-based, mission critical platform for organizations planning their futures around reliable architectures. Oracle Application Server 10g Release 3 has extended the State Replication features of previous releases to provide the most scalable **and fault-tolerant** session replication framewor with special empahsis on guarantied services, reliability, ordering, fragmentation and group membership.

This paper addresses recommendation and best practices surrounding State Replication for a stateful application at the Servlet/JSP layer (HTTP Session replication[1]). It also demostrates how Oracle Applicatiion Server 10g R3 offers practical linear scalability for stateful configurations and explains the different replication options available. We also walk through all of the major configuration parameters and discuss the impact and benefits of different replication settings on Availability, Performance and System Resources.

Our tests and observations support the conclusion that enabling Replication adds minimal overhead, is a fully scalable solution, and should be enabled in all cases where an application's session data is deemed critical.

[1] Although state replication for Stateful Session Bean is also available and uses the same replication framework as HTTPSessions in OracleAS 10g R3, it is not addressed nor analyzed in this doc

## INTRODUCTION

The decision to enable replication on a stateful application involves considering the following questions:

1) What is the impact on my application if session state is lost?
2) How does state replication protect against loss of state and increase the availability of my environment?
3) Will enabling state replication affect the performance of my application?
4) Will enabling state replication have an adverse effect on my system resources?

Regarding the first question above: The impact of losing session information will depend on the nature of the application. Sessions often are used to keep track of a specific user's progress through a series of pages or decisions. Loss of this data will mean a reinitialization of all session state for all users.

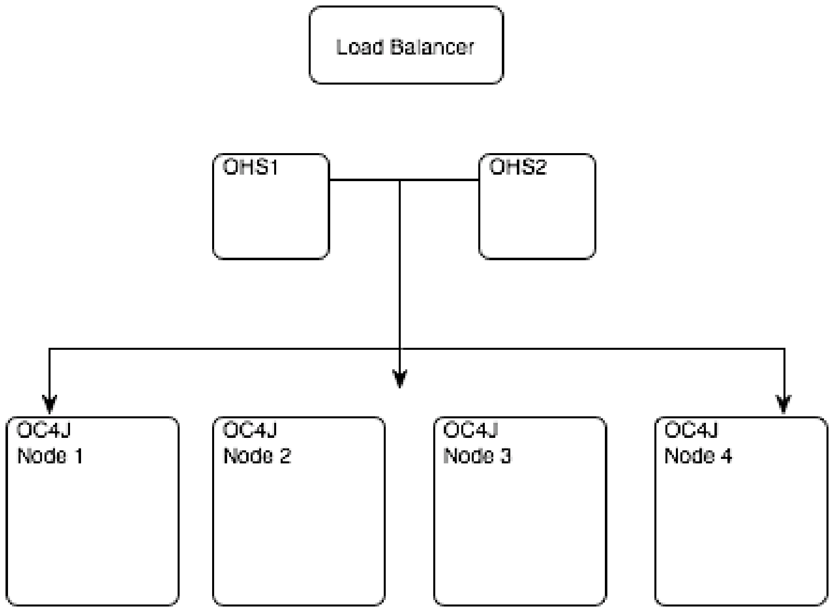
The other questions posed above are the ones we address in this paper. That is, what is the overall cost of replication and what are the effects of the different replication options? How will this affect an existing application in terms of performance, availability and resources?

The first part of this paper introduces a reference replication topology and defines some basic terminology. Then we discuss the replication options with a short discussion of their effect as well as our recommendations. The latter part of this paper discusses some of the specific tests performed and presents some graphs of our own test data which support the recommendations.

Scalability recommendations and observations follow, especially pertinent to high-traffic or high-replication environments. Finally, the appendices include more detailed information on our specific environment as well as details on our JVM configuration.

## REFERENCE REPLICATION TOPOLOGY

Before proceeding to discuss the specifics of a J2EE Clustered Replication environment, we introduce a typical topology for further reference.



## Topology Components and Terminology

### Load Balancer

We will use a Load Balancer to direct client requests to one of our OHS servers. This will normally be configured with an HTTP-based health monitor to determine which of the OHS Servers are available to serve requests.

### Oracle HTTP Server (OHS)

The OHS Server accepts HTTP requests and then forwards the Servlet request to one of the available OC4J Servers in the OPMN Topology.

### Oracle Containers for Java (OC4J)

The OC4J Server accepts the request from one of the OHS Servers, processes the request and returns the result to OHS.

An OC4J installation may be running one or more OC4J Instances. An Instance may include 1 or more JVMs. In most cases (the notable exception being the colocation tests) we ran with 1 OC4J instance with 1 JVM.

### OPMN-based Topology

All of the OHS Servers and OC4J Servers are configured in a Multicast OPMN Topology. Each of the OHS Servers can route to any of the OC4J Servers in the diagram above. Membership in the topology is dynamic. If an OC4J Server is taken down, it is removed from the OHS routing table. More OC4J Servers can also be added by configuring OPMN parameters in the **opmn.xml** file.

### Replication Cluster

A Replication Cluster is an OC4J clustering mechanism orthogonal to the OPMN topology. In Oracle Application Server 10g Release 3 replication of state for J2EE applications can be enabled both at the application and the container level thus providing a more granular control of the replication scope.The OC4J Session replication clusters and the OPMN topology are generally distinct except that:

1. A Replication Cluster can be a subset of an OPMN topology.
2. The OPMN topology can optionally be used as a discovery mechanism for a Replication Cluster.

The OPMN configuration is used for dynamic routing of requests and dynamic topology membership. The Replication cluster is used for propagating session state. These two groups (OPMN topology and Replciation Cluster) use two distinct communication channels.

## REPLICATION CONFIGURATION OPTIONS

After making the initial choice to replicate session state, there are still a variety of options to consider in how the replication is performed and in what circumstances. Here we review some of the basic Replication Topologies and Configuration Options.

### Multicast vs. Peer-to-Peer

When configuring replication, three different network configuration options are documented: Multicast, Dynamic Peer-to-Peer and Static Peer-to-Peer. The differences among these three are:, briefly:

***Multicast*** – Uses IP Multicast for replication. Each replicating host is configured with the same multicast address. All hosts are then part of the same Replication cluster.

***Static Peer-to-Peer*** – Each host is configured with the IP address of one other peer. As long as there is a chain of peers linking all hosts together, then a Replication cluster is established. This unicast communication is used both for discovery and for replication.

***Dynamic Peer-to-Peer*** – In this configuration, Peers are discovered via the OPMN Topology. This is only used for discovery. After the discovery phase, the replication peer-to-peer cluster manages replication in the same manner as the static peer-to-peer configuration described above.

There are trade-offs involved in choosing a replication clustering mechanism. The introduction of Peer-to-Peer clustering was introduced in 10.1.3 to increase the reliability (over UDP based multicast) of communications among the JVMs. The recommendations in this paper apply to a peer-to-peer environment.

In stand-alone OC4J installation, the only peer-to-peer option available is static. Otherwise, if OPMN is available (as in an Oracle AS instance installation), a dynamic peer-to-peer cluster will be easier to configure and maintain and is the recommended option.

If only static peer-to-peer is possible, the nodes should be configured in a ring model so as to have a complete cluster even if a node is unavailable. For example, in a  4 OC4J node configuration (A,B,C,D) the nodes should be configured so that B is configured with A as a peer, C with B as a peer, D with C as a peer, and A with D as a peer.

## Distributed vs. Co-located JVMs

Replication can occur either to another JVM on the same machine and instance or to a remote JVM on another node. The boolean parameter **allow-colocation** can be set to allow or disallow local replication. The advantages and disadvantages of replicating to a local JVM are outlined below:

### Response Time

The difference between making a local replication request and one across the network can be one of milliseconds, depending on the network speed. In most cases this will be an insignificant extra overhead to the application.

In the case of default, asynchronous replication, our tests showed that *the extra latency in the request time stays fixed even as the amount of replicated state increases.* That is, in an asynchronous model, the only extra delay is in communicating with the remote replication JVM, not in the actual transmission of state.

### Availability

Replication provides greater Availability and protection against failures. Although a JVM on the same machine protects against an application or JVM failure, it provides no protection against failure of the entire node, if the JVM is only replicating locally.

### System Resources

Enabling Replication to another JVM effectively doubles the memory resource requirements. Each JVM maintains its own session state as well as that of any other JVMs which are sending session state to replicate. If both JVMs are on the same machine then adding additional memory for a new JVM may not be feasible.

A remote JVM, however, will add to the network bandwidth requirements. *This scales linearly with the amount of state that is being replicated* since all of the state must be transported over the network. The impact of replication network traffic on performance is described in the “Response Time” section above, with the caveat of course that network bandwidth is not saturated.

## Number of Replication Targets or Replication Quota

The number of other JVMs to replicate to is controlled by the parameter **write-quota** with a default value of 1. Considerations of the optimum value for this parameter should take the following into account:

### Response Time

In asynchronous replication mode, state is sent to a remote JVM but there is no wait for acknowledgment. In this case, sending state to multiple remote JVMs does not noticeably increase the response time of a replication request. In the case of comparing a write-quota of 1 and 2, *no appreciable difference was found in the response time.* Since the session state is sent asynchronously, it can be quickly sent to any number of external JVMs.

In synchronous replication, the application waits for acknowledgment from at least one node, rather than all nodes which are receiving replicated state. So the addition of more replication targets also does not add to the total response time.

This of course assumes sufficient system resources. More detail on this is in the 'System Resources' section below.

### Availability

More replication targets increases the availability of the system. If the original JVM fails, there is more than one other JVM which still holds the session state. On the other hand, this can also be a matter of diminishing returns. If the write-quota is set to 1 and the replication is occuring to a JVM on another node, it would take *two* independent node failures for all session state to be lost.

In practice, this is an uncommon enough occurrence that a **write-quota** of 1, with the target set as a remote node (**allow_colocation**=false) should be sufficient for most purposes.

### System Resources

A replication request produces both outbound network traffic, as the state of local JVMs is sent to remote nodes, as well as inbound traffic, as remote state is received locally. The amount of total traffic is proportional both to the amount of state being replicated as well as the number of nodes involved in the replication.

The amount of inbound traffic during a specific time period, for example, can be roughly calculated as:

$$I=Reqs*Wq*St$$

Where I is the total traffic, Reqs is the number of replication requests per application instance during that time period, Wq is the write-quota and St is a factor of the average size of the replicated Session state. That is, on a busy system with a lot of replicated state, increasing the write-quota from 1 to 2 will double the traffic and may or may not be acceptable.

Likewise, a JVM managing double the amount of session state is a JVM with potentially increased memory requirements and more memory management overhead – in the form of increased garbage collection activity for example.

It should be emphasized that these considerations should not come into play at all in a system unless resources are otherwise constrained.

## Asynchronous and Synchronous Replication

The default replication mode is asynchronous. The difference between asynchronous and synchronous is in whether the application waits for an acknowledgement from at least one replication target after the data has been sent. In the asynchronous case, the application sends the data and continues with application processing concurrently.

Again, we summarize the impact of this extra acknowledgment.

### Response Time

Waiting for the acknowledgement does add to the total response time. This extra time will not be large for small bits of state going across a fast network. In the cases of an application with small, fast servlets, the effect of the extra overhead of synchronous replication, on the order of milliseconds, may as much as double the total response time. For larger more complex applications, the extra milliseconds to wait for an acknowledgement may be negligible.

### Availability

The advantage of synchronous replication is that it protects against inconsistencies in the session when a failure happens before session state has been replicated to another node.

Thus enabling synchronous replication does increase reliability, but only for a small window. Both asynchronous and synchronous replication wait for an initial acknowledgement from the remote host that it is available to receive data. So the remote host must disappear in the window of time after which the remote host has made an initial acknowledgement but during the transmission of data, while it has not been completely received.

### System Resources

Choosing to enable synchronous replication does not have an appreciable effect on any system resources. The CPU time may be seen to be slightly *lower* for synchronous replication since a small extra wait time has been added to the application.

For most cases, the default asynchronous replication should provide sufficient availability. Enabling synchronous replication does detract from the performance of the application while covering a small availability window. Whether the trade-off should be made will be an application-dependent decision.

### When to Replicate

The default parameter for replication frequency is OnRequestEnd. This specifies that replication occurs at the end of the request. The other options are OnAttributeModify and OnShutdown. The latter option will only replicate if the application is normally shutting down and thus will not protect against most failures.

For all of our testing we have employed the option OnRequestEnd. From an application perspective, the benefit of session replication is to hold and maintain state between several independent requests. Replicating more frequently will have a performance impact but, more importantly, may not produce data that is consistent or usable since it was produced mid-request but now must also be valid at the initiation of a request.

## REPLICATION TESTS

The following sections delve into a bit more detail on the actual tests performed, results obtained and motivation for the recommendations. These sections parallel those in the previous section.

### Multicast vs. Peer-to-Peer

Initial tests showed the variability of multicast communications to be not as relaible as peer to peer for our testing purposes. That is, the distribution in latency for a series of replication requests affected other variables we were interested in exploring. And so, beyond a few initial tests, we chose to perform all further tests using Peer-to-Peer replication.

### Distributed vs. Co-located JVMs

The tests with colocated JVMs all involved a total of two JVMS, configured to replicate to each other. This involved the following configurations:

1) Two Instances on the same machine, each instance with one JVM
2) One instance, configured with two identical JVMs
3) Two Instances, each on their own machine, each instance with one JVM.

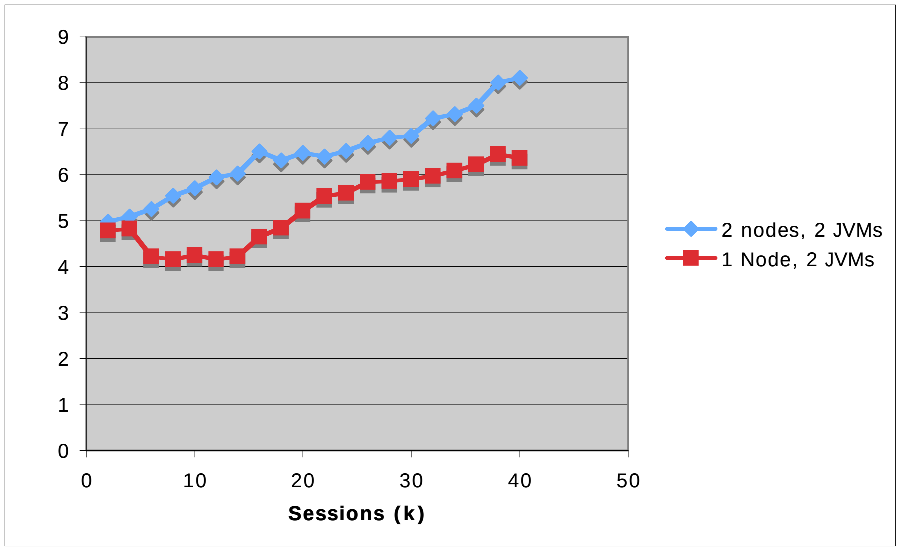
The immediate impact of having two JVMs share the same machine is that this situation is more likely to saturate the resources on that particular machine. This assumes that the memory allocated to each JVM remains the same.

Regarding Availability, configuration 2) provides proteciton at the JVM level but does not provide any protection against OracleAS instance nor node failure. On the other hand, this configuraiton is very easy to manage by OracleAS (automatic port allocation is done for the different JVMs by the instance) and provides a very dynamic way to scale vertically (inside a node) . Configuration 1) will protect against instance failure but will not protect against machine/network failure.

It was observed that the CPU requirements doubled when having two JVMs on the same machine, as might be expected. Placing the JVMs in separate instances requires slightly more resources. On a constrained system we expect that this will have a strong adverse impact on performance.

What about an unconstrained system? In that case, we would expect the difference to be that between local replication and replication across a network.

The graph below shows the response time, in milliseconds, of two different configurations. In both these cases, all of the machines involved had sufficient memory and CPU. In one case, however, the two JVMs are replicating to each other locally. In the other case, the JVMs are replicating across a fast network.

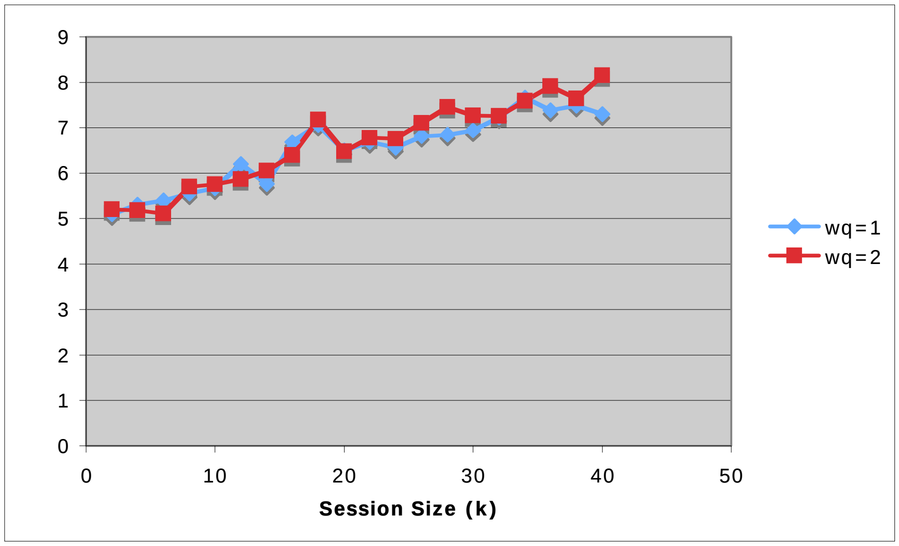


The horizontal axis in the graph demonstrates the effect of increasing the amount of state being replicated. Both configurations show an increase in response time, presumably due also to the greater amount of work that the test servlet must do to process more state. But the extra amount of replicated state does not increase the amount of *extra* time required by the servlet.

It should be emphasized that these tests were performed on a system with sufficient network bandwidth. The amount of bandwidth used does increase linearly with the amount of state being replicated (refer to the previous section on Replication Targets for rough calculations) and this will undoubtedly affect the performance if the network is becoming saturated. **It should be noted that response time barely doubles for a configuration with almost 5 times the size and number of sessions. This demonstrates a good workload adapation by the replication framework in such a topology**

### Quota or Number of Replication Targets

Our tests found no noticeable additional overhead from replicating to multiple targets other than the potential impact on network bandwidth. Since our tests were performed on a system with sufficient resources, we saw no discernible difference at all, as shown in the graph below.



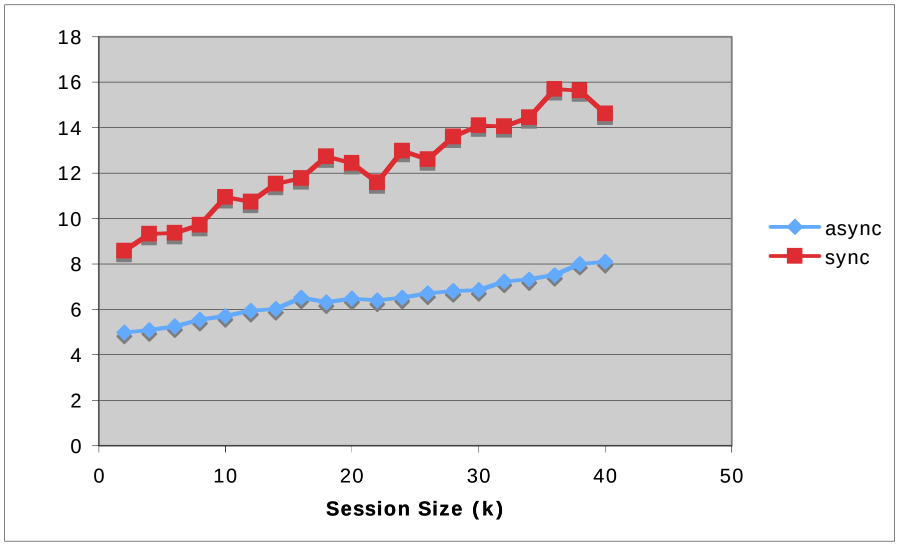
The test represented by the graph above was performed in an environment with four different JVMs all configured with replication. In one case, each was configured with a **write-quota** of 1, replicating state to one other JVM. In the second case, the **write-quota** was set to 2, each JVM replicating to two other JVMs.

The response time was also measured across a range of total replicated state size. As can be seen above, within the bounds of normal variability in response times, the two configurations had no impact on the total response time of the test servlet. This is due to the fact that replication happens in paralell to the different members, hence we can increase the realibility of the replication cluster (by increasing quota) without a major impact in perfromance

## Asynchronous and Synchronous Replication

The difference between synchronous and asynchronous replication is in whether the originating JVM waits for acknowledgement that all data has been received by at least one other JVM before proceeding. This also explains why increasing write-quota is fairly benign – in that case, synchronous replication only waits for acknowledgement from the first JVM to receive the data.

The wait time for the extra acknowledgement, however, will increase the response time of the application. The amount of extra wait time will depend on the application. In the chart below, we show response time as a function of total state being replicated.



Enabling synchronous replication does add to the total response time as above. However, this is on the order of milliseconds per request. In the case of our test servlet which does very little other than replicate, this effectively doubles the response time.

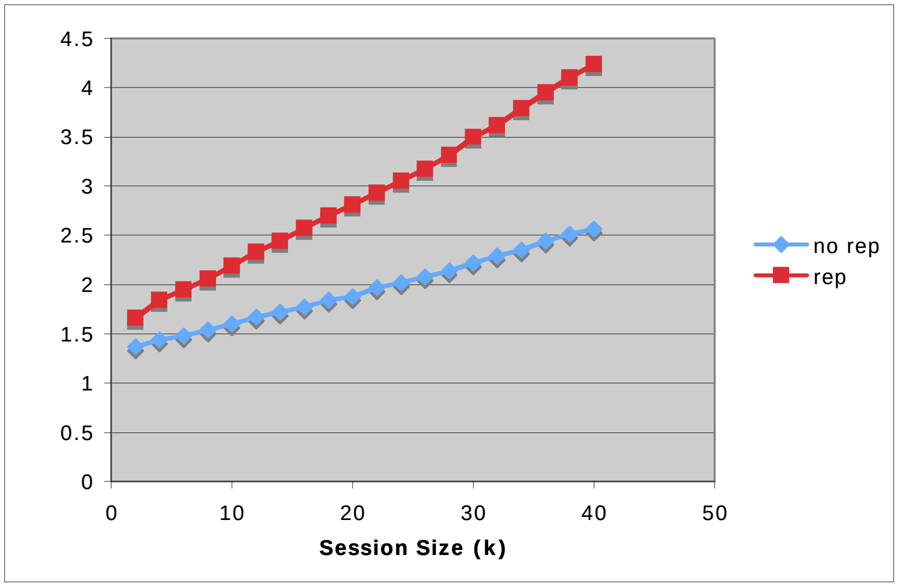
With the increase in the size of session state being replicated, the extra amount of time increases.  In general, the major driver for the impact of syncrhonous mode on the perfromance of the system is the replication trigger (i.e. depending how many times the session is replicated in an application, and how long the application takes to process that info) the impact may be higher as the wait for acknowledge will happen more often and the wait period will increase.

## SCALABILITY RECOMMENDATIONS

The best practices involved in creating an appropriate replication topology will, to some degree, be application dependent. Using our reference topology introduced earlier we can make some observations about the impact of enabling replication, recommendations on scaling out replication topologies as well as some remarks about availability.

### The cost of Replication

The graph below compares the response time of a small test application both with replication enabled and replication disabled.



The graph above represents the measured response time of a small servlet. In this case, we have used the default asynchronous replication, a write-quota of 1 and peer-based replication.

The response time is a measured average of five concurrent clients over a sufficient period after which the system has stabilized. The horizontal axis represents the amount of replicated state. It is recommended not to use session sizes above a few 100kbs since this increases the perfromance impact and also the amount of data loss in all cases.

This also represents a system with sufficient resources in terms of Memory, CPU and bandwidth. The extra time added by asynchronous replication is processing time.

Although the extra cost of replication does increase with more session state, the average increase in processing time, especially for small amounts of state is on the order of milliseconds per request.

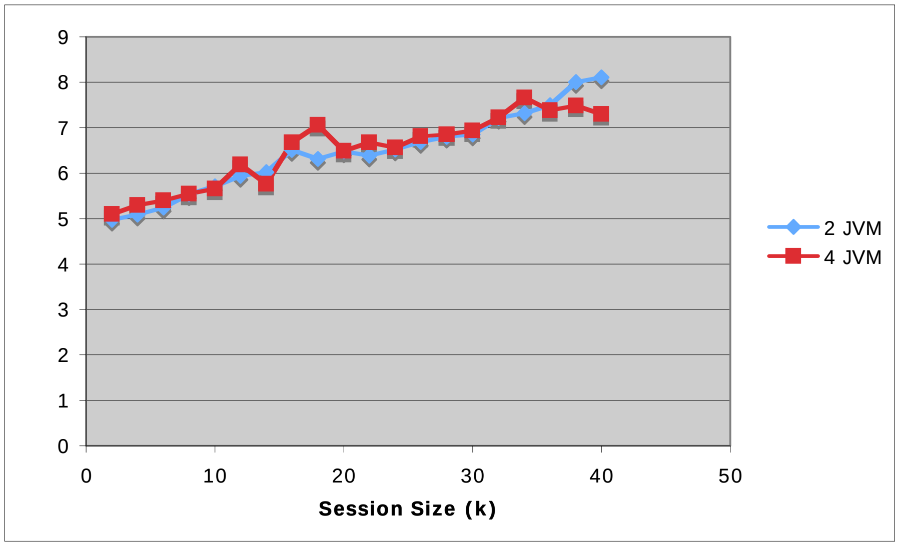
## Scaling with Replication Pairs

Scaling up an application involves the addition of more resources to handle the load in a manner that is repeatable and ideally with no immediate limits.

With a write-quota of 1, each JVM is only replicating to one other JVM. This naturally suggests the idea of replication pairs – pairs of JVMs which are replicating to each other.

In the reference topology outlined in this paper, we have OC4J1 and OC4J2 replicating to each other. We can continue to add JVMs in pairs. Adding OC4J3 and OC4J4, for example, will create a new pair which replicate to each other.

The graph below shows the response time in a topology with 2 replicating JVMs and a system with 4 JVMs.



In an unconstrained system, we would expect the response time of an application to remain the same when more resources are added. If the response time goes down, then the system was constrained after all. If the response time goes up, we expect that the addition of more resources has introduced overhead somehow.

## Failover Planning

The greatest advantage of session replication is in protecting against outages. These outages can take the form of process failures, or hardware or network failures.

Here we discuss some possible outages and how replication fails over the state in each case:

### JVM Failure

If a JVM crashes or is otherwise unavailable, it will be removed from the OPMN Topology and also removed from the Replication cluster. Surviving JVMs will reconfigure themselves to find a new replication destination.

Finding a new replication partner is best-effort. That is, if write-quota is set to 2 but only one other surviving JVM exists, then replication will only occur to one destination. Likewise, if a JVM is the only surviving JVM it will continue to function and  process data even though there is no replication destination available. That is, the system will continue to function and will not hang or wait for the designated number of replication partners.

If a previously unavailable JVM rejoins the cluster, the cluster will again reconfigure itself to enable the requested number of replication destinations if this is possible. In this model a node a JVM acts as a replication master that triggers replication to meet quota as other members become avasilable. To trigger replication quota fullfillment, it s necessary that the replica of the session is active (i.e. has been accessed at least once in that node) by a user request

### OC4J Node Failure

Tha failure of an entire machine, instead of just the OC4J processes, is similar to the failure of just the JVM process. In both cases, the surviving nodes will reconfigure themselves as described above.

In the case of a node failure, however, the wait time is longer before the node is ejected from the Replication cluster. In this case the wait is on a TCP timeout of the open connection between the now-dead node and the surviving nodes.

### OHS Failure

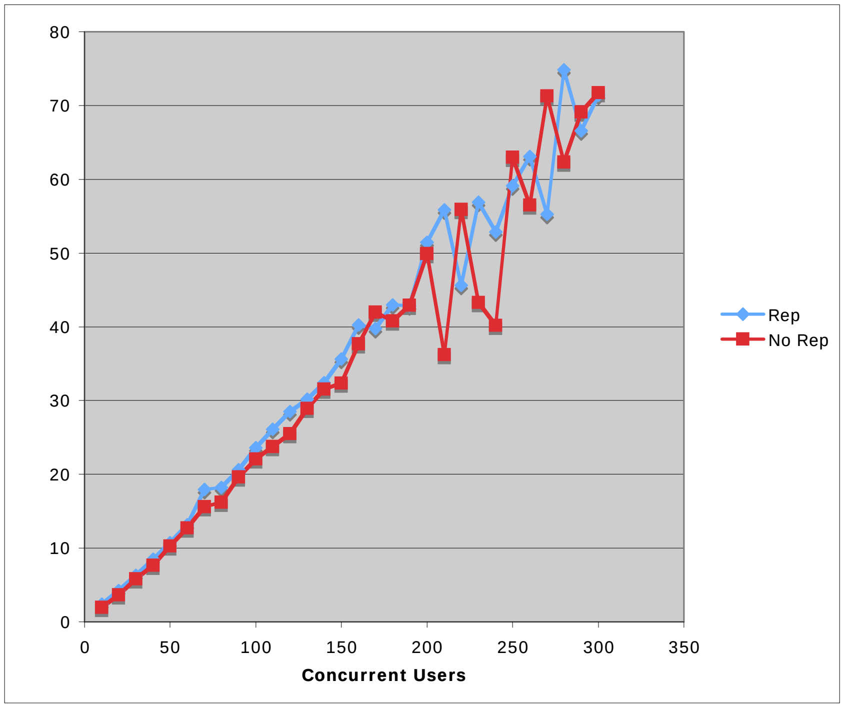
Since OHS is the point of entry for the system, clients detect a failure of either the process or the node immediately. In our reference architecture in this paper, we have two OHS nodes. If one becomes available, the surviving OHS node is able to perform the same routing as the unavailable one.

## Other Scalability considerations

### High Load Scenarios

The graph below shows the response time as we increase the load on the system – both without replication and with replication enabled.

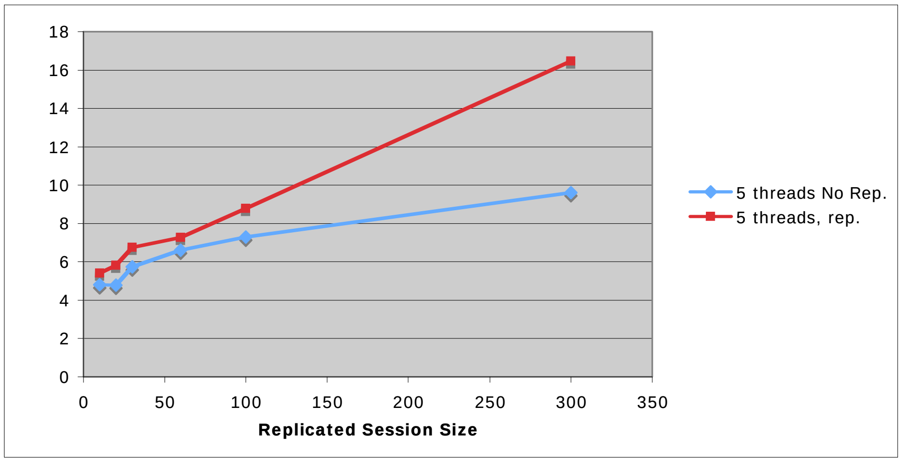
The horizontal axis represents the number of concurrent users, each making small (1k) session requests. The vertical axis is the response time in millseconds.



Although our system appears to become erratic above 200 concurrent users, due to the system becoming overloaded, there is no appreciable difference in the response time of the application with or without replication enabled. Likewise, at high loads, the response of a replication-enabled system does not reveal any scalability concerns. This is as clear a demostration we were able to see that the OracleAS 10g R3 replication framework is highly scalable.

### Large Session Sizes

How does replication scale with larger session state sizes?



The graph above shows the response time as a function of session size. Above a state size of around 100k, the response time increases as a function of the state size, rather than being a fixed overhead. At this point, the network bandwidth is becoming heavily used but the increase appears to be linear. Still, response times are increased in the order of fractions of a second for session increases of several hundrd K, and there is an order of seconds impact difference between a session replciation enabled system and a system where replication is not being used.

## CONCLUSION

The costs of enabling J2EE State Replication have been outlined here in this paper. From our tests and observations, we have determined that the additional cost of enabling asynchronous replication is minimal both in terms of impact on performance and required system resources and thus the overall scalability of the replication system is extremely good.The gain in application availability justifies the extra overhead for applications where it is relevant.

The parameters recommended for most cases are:

Asynchronous replication - Although synchronous replication offers greater consistency guaranties than asynchronous, this may be a case of diminishing returns.

Distributed JVMs – Only JVMs on different physical machines can protect against network or machine failures. Dependeing on the scalability model required in a system (horizontal vs. vertical) either one or the toher may be acceapble with the trade offs describe in this document

One Replication Target – Although multiple replication targets can offer greater protection without significant overhead, one replication target should be sufficient for most cases.

Peer Replication – Peer replication can offer greater control for setting up a replication topology. Coupled with Dynamic OPMN-based discovery, this is also a manageable solution.

## APPENDIX A: CONFIGURATION DETAILS

### Hardware Configuration

[NOTE: Drivers and Application Servers were identical unless otherwise noted]

- HP Integrity rx2620 servers
  - 2 x 1.6 GHz Single Core Processors
  - 12 GB RAM
- 2 x 146 GB internal disk drives
- HP-UX 11iV2 operating system software
  - Patch Bundles: March 2006 Quality Pack; June 2006 HW Enablement
  - Java 1.5.0.03
- Networks:
  - MP (Maintenance Port) 100TX
  - Public 1GB
  - Private 1GB (Dedicated Apps-Tier Load Balancer network for all non-replication communication)
  - Private 1GB (Dedicated Replication network for Apps-Tier)

### HPUX Configuration

For most of the tests, HPUX 11v2 was used on all servers. Toward the end of this investigation, one of the application servers was upgraded to HPUX 11v3., and some baseline tests were rerun. The results showed that there was no difference in performance between 11iV2 and 11iV3. The patch requirements for 11iV2 are listed below. No additional patches were required for 11iV3. Also listed are the kernel parameters, which are same for both 11iV2 and 11iV3.

HPUX 11iV2 Patches

| | |
|---|---|
| PHCO_34944 | pthread library cumulative patch |
| PHKL_34032 | ksleep cumulative patch |
| PHSS_34444 | assembler patch |
| PHSS_34445 | milli cumulative patch |
| PHSS_34853 | Math Library Cumulative Patch |
| PHSS_34858 | linker + fdp cumulative patch |
| PHSS_34859 | Integrity Unwind Library |
| PHSS_35045 | Aries cumulative patch |

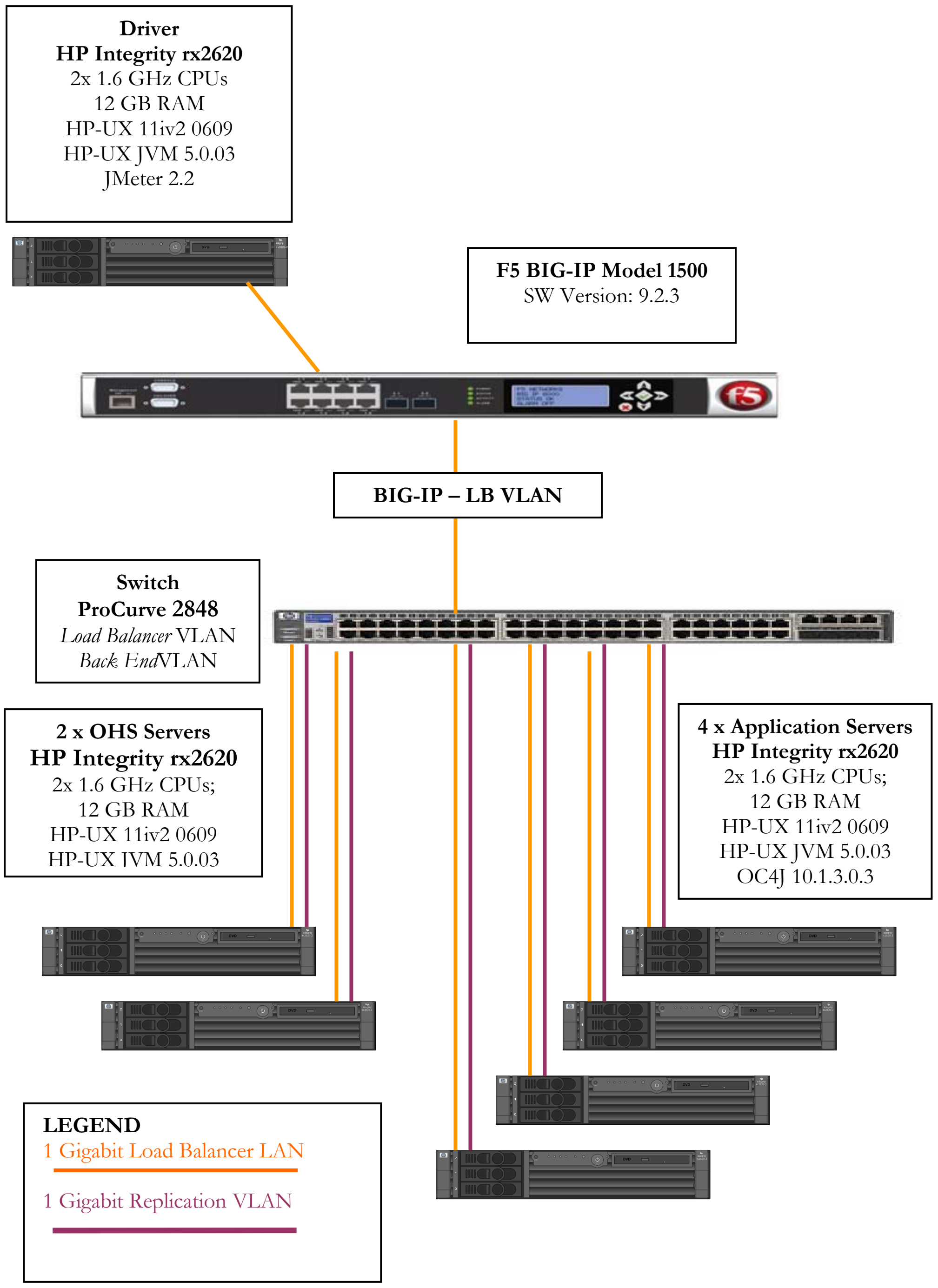
PHSS_35055 aC++ Runtime (IA: A.06.10, PA: A.03

*HPUX Kernel Parameters*

| Parameter Name | Value |
|---|---|
| cmc_plat_poll | 15 |
| create_fastlinks | 1 |
| dbc_max_pct | 8 |
| dbc_min_pct | 8 |
| default_disk_ir | 1 |
| fs_async | 1 |
| hfs_max_ra_blocks | 20 |
| hfs_max_revra_blocks | 20 |
| hfs_revra_per_disk | 256 |
| max_async_ports | 768 |
| max_thread_proc | 2048 |
| maxdsiz | 4294963200 |
| maxfiles | 32768 |
| maxfiles_lim | 32768 |
| maxssiz | 401604608 |
| maxtsiz | 1073741824 |
| maxuprc | 3277 |
| maxvgs | 80 |
| msgmap | 5122 |
| msgmax | 32768 |
| msgmnb | 65536 |
| msgseg | 20480 |
| msgssz | 128 |
| msgtql | 5120 |
| nfile | 65536 |
| ninode | 8192 |
| nkthread | 16384 |
| nproc | 8192 |
| npty | 200 |
| nstrpty | 200 |
| nswapdev | 25 |
| o_sync_is_o_dsync | 1 |
| scsi_max_qdepth | 8 |
| semmni | 4096 |
| semmns | 8192 |
| semmnu | 4092 |
| semume | 512 |
| shmmax | 2000000000 |
| shmmni | 520 |
| shmseg | 512 |
| STRMSGSZ | 65535 |
| swapmem_on | 1 |

| | |
|---|---|
| swchunk | 8192 |
| tcphashsz | 32768 |
| vps_ceiling | 64 |

## APPENDIX C: DIAGRAM OF ACTUAL TEST ENVIRONMENT



## APPENDIX D: JVM CONFIGURATION

The HP Java 1.5 was used for all of the testing. However two different patch releases were used. Most of the testing was done using the version that is shipped with AS 10.1.3.x:, java version "1.5.0.03"

However a series of baseline tests were run using the latest version of the HP Java, java version "1.5.0.08". The tests using the newer version showed similar performance characteristics.

In addition to varying the JVM version, we also varied the JVM parameters. Most runs were performed with the following set of parameters:

> *-server -Xmx3000m -Xms3000m -Xmn2000m -XX:PermSize=48m -Xverbosegc:file=/tmp/gcfile -Djava.security.policy=$ORACLE_HOME/j2ee/project/config/java2.policy -Djava.awt.headless=true -Dhttp.webdir.enable=false*

However test runs were also performed with the full set of performance related parameters that had been used for SpecJAppServer2004 benchmarks (see

http://www.spec.org/osg/jAppServer2004/results/res2007q1/jAppServer2004-20070130-00054.html

Perhaps because there were sufficient system resources for all runs, these additional parameters did not result in any performance improvements.

There was sufficient memory so as not to produce any distractions for memory management issues such as JVM Garbage collection. Tuning for a resource constrained environment was not the focus of these tests

## APPENDIX C: CLIENT, SERVLET AND JVM CONFIGURATION

The Servlet used for the test environment provided three functions:

1) Create state
2) Process state
3) Destroy state

A typical call to the servlet was of the form:

http://host:port/Session/Session?mode=initialize&number=2&size=5

This example would create 2 session objects, each 5k in size. The process function loops through all the session objects, modifying the value of each one. The final function releases all of the session objects.

The calling clients were emulated using Apache's Jmeter. Typical runs were 5 threads each implementing the functions above, processing 4 times before destroying the state. There was no think time. The measured response times presented in this paper were gathered by averaging the reported output values from Jmeter .jtl files over the course of 10k-20k runs per thread. The resulting average is a mean value, calculated after discarding the first 20% of the run (to correct for any initialization issues and get a steady state value).

The JVM parameters used in these runs was:

*-server -Xmx3000m -Xms3000m -Xmn2000m -XX:PermSize=48m -Xverbosegc:file=/tmp/gcfile -Djava.security.policy=$ORACLE_HOME/j2ee/project/config/java2.policy -Djava.awt.headless=true -Dhttp.webdir.enable=false*

This was sufficient memory so as not to produce any distractions for memory management issues such as JVM Garbage collection since that was not the focus of these tests. The output gc file was used to verify that this was the case.

**Oracle J2EE State Replication Guidelines and Best Practices**
**October 2007**
**Author: Richard Delval**
**Contributing Authors: Pradeep Bhat, Fermin Castro, Bill Cortright**

**Oracle Corporation**
**World Headquarters**
**500 Oracle Parkway**
**Redwood Shores, CA 94065**
**U.S.A.**

**Worldwide Inquiries:**
**Phone: +1.650.506.7000**
**Fax: +1.650.506.7200**
**oracle.com**