OpenLRS Technical Guide

OVERVIEW

OpenLRS is a secure, standards-based, standalone Learning Record Store. OpenLRS was built to fill the need for a high i/o storage mechanism for an open learning analytics environment, as displayed in the diagram on the right. Built on a scaleable architecture, using modern web technologies, OpenLRS provides the fast reads and writes necessary for a dynamic analytics environment.

This document includes the following sections:

- High-level design (including overview of the OpenLRS web application and tiered storage mechanisms)
- Small-scale pilot or demo environment deployment
- Small-scale pilot or demo environment performance baseline
- Large-scale production environment deployment
- Large-scale production environment performance

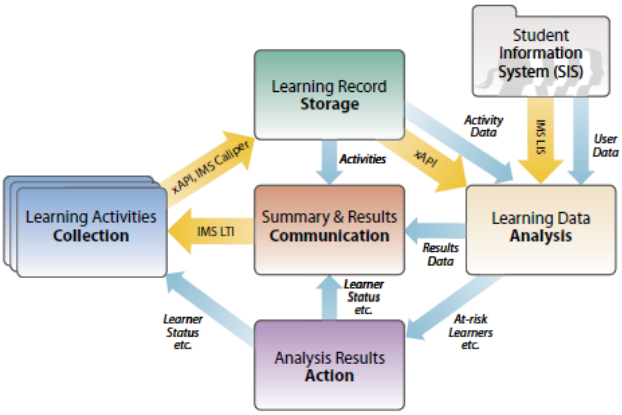


Diagram: An Open Learning Analytics environment

HIGH-LEVEL DESIGN

OpenLRS Web Application

The OpensLRS web application provides the xAPI interface, secures the API, and manages the retrieval, storage, and indexing of learning records. A two-tier storage solution capability will be built-in, in order to allow the ability to leverage separate storage solutions, one that provides high speed writes and another that provides high speed reads. The web app will use a pluggable data access layer that allows flexibility in deciding which backend storage solutions to use. This is important because the response times and scalability will be heavily dependent on the storage and indexing solutions used. Security via xAPI HTTP Basic and OAuth 1.0a will be provided.

Data Access Layer

As mentioned previously, the data access layer will be pluggable, meaning that backend storage solutions can be swapped out as needed depending on cost/performance/infrastructure needs. The two-tier storage capability will allow for separate storage products to be used for "writes" (Tier 1) vs. "reads" (Tier 2), however, it will still be possible to use just a single storage product for both "reads" and "writes", if that is desired. For example, one could use MongoDB for Tier 1 and MySql for Tier 2, or just as well use Oracle DB for both Tier 1 and Tier 2.

Connector

If separate storage solutions are used for Tier 1 (writes) vs. Tier 2 (reads), then it may be necessary to use a "connector" that can transfer the data from Tier 1 to Tier 2.

Technology Stack

- Java 7
- Spring Boot

Tier 1 Storage

Key Critieria

- high volume
- high speed writes
- transactional accuracy

The Tier 1 storage for the learning records will initially be Redis.

Redis Features

- outstanding performance
- in-memory only (or persist to disk)
- master/slave async replication

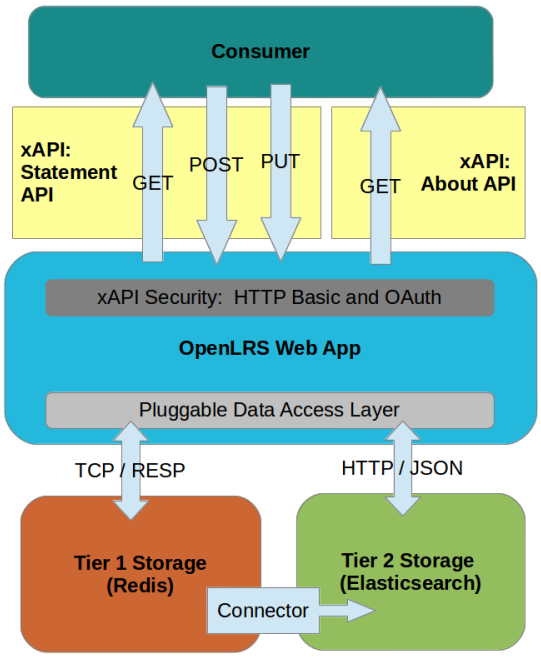
Redis Master-Slave

- Full Redis replication documentation
- Full Redis sentinel documentation

A Note About Redis Cluster

Ultimately redis cluster may be the appropriate solution for HA Redis with OpenLRS. At this time, however, redis cluster is still in alpha and has not been deemed appropriate for production use. Note some of the concepts mentioned above will continue to be applicable if and when redis cluster is ready.

http://redis.io/



Tier 2 Storage

Key Criteria

- high volume
- high speed reads
- indexing support

The Tier 2 storage used for the learning records will initially be Elasticsearch.

Elasticsearch Features

- horizontally scaleable
- highly available
- very fast
- build on Lucene (for indexing)

http://www.elasticsearch.org/

Use Cases

- save a statement (POST)
- retrieve a statement (GET)
- retrieve a group of statements (GET)
- retrieve "about" info (GET)

API

xAPI About

https://github.com/adlnet/xAPI-Spec/blob/master/xAPI.md#aboutresource

xAPI Statements

https://github.com/adlnet/xAPI-Spec/blob/master/xAPI.md#stmtapi

xAPI Security

- HTTP Basic Authentication
- OAuth 1.0a Authentication

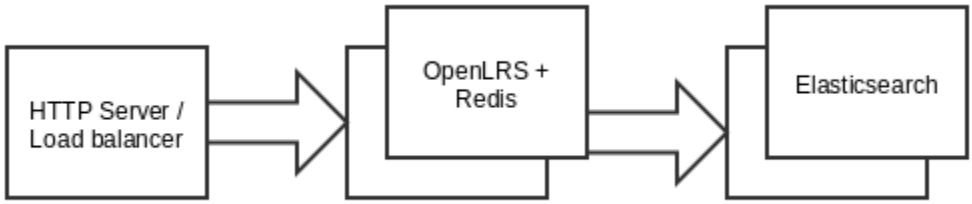
https://github.com/adlnet/xAPI-Spec/blob/master/xAPI.md#security

SMALL-SCALE PILOT OR DEMO ENVIRONMENT

The following recommendations are for a long term demo/test environment or small (less than 10 course section) pilot of OpenLRS.

Considerations

- Minimal upfront and maintenance costs
- Acceptable but not production quality performance and redundancy
- Potential to transition the storage tier to production (i.e. the OpenLRS + Redis instances are expendable but you might want to keep the Elasticsearch instances - more importantly the data on them - long term)



Component	Instance Count	Recommended Specs	Notes
HTTP Server / Load balancer	1	N/A	Apache HTTP, Nginx, Amazon ELB or similar.
OpenLRS + Redis Instances	2+	JDK 7 RAM: Minimum 8 GB / CPU: Quad core / Disk: Minimal storage needs	
Elasticsearch instance	2+	JDK 7 RAM: Minimum 16 GB / CPU: Minimum dual core / Disk: Minimum 100 GB (SSD preferred)	Hardware recommendation from elasticsearch

OpenLRS + Redis Instances

In a true production environment OpenLRS and Redis would be run on separate instances. Additionally you would likely want to have a clustered Redis with automatic sharding and persistent message queues. However, for a this type of environment that kind of investment is probably cost and time prohibitive. Instead the recommended approach for this environment type is simply to run a separate Redis server along side each OpenLRS instance. At least two OpenLRS + Redis instances are recommended for failover purposes.

In terms of hardware, memory and CPU are most important for the OpenLRS instances. 8 GB of memory is recommended as it will be shared by both Tomcat and Redis. Also a multi-core processor is recommended to allow for handling of a scaling of concurrent requests in Tomcat (i.e. increasing the number of maxthreads). Disk needs are minimal (logging).

Elasticsearch Instances

Since you may want to keep data gathered from a pilot long term the Elasticsearch instances should be considered production quality. Elasticsearch provides their own hardware recommendation here. Memory and disk are most important for Elasticsearch and that is especially true for OpenLRS where the vast majority of operations will be writes.

At least two Elasticsearch instances are recommended and each node should be configured to run within a cluster. Options for configuring the cluster can be found here.

SMALL-SCALE PILOT OR DEMO PERFORMANCE

Summary

The following are the results of baseline performance tests for OpenLRS. Performance tests were executed in a development environment using JMeter test scripts developed by the Apereo Learning Analytics community (https://github.com/Apereo-Learning-Analytics-Initiative/LRS LoadTest).

Overall OpenLRS performed well. In capacity tests OpenLRS maintained an average response time of less than two seconds per transaction. In stress tests, OpenLRS response times increased proportionally with concurrent users with an average response time of around five seconds per transaction.

Test Objectives

- Establish a performance baseline for OpenLRS.
- Validate software design and architecture choices

Test Conditions

Software

- Ubuntu 14.04
- OpenLRS (TwoTierRepository configuration with embedded Tomcat 7)
- Redis 2.8.13
- Elasticsearch 1.3.2
- OpenJDK 1.7.0_65

Hardware

- Memory: 16 GB
- Processor: Intel® Core™ i7-4900MQ CPU @ 2.80GHz × 8
- OS-type: 64 bit
- Disk: 240 GB

Other considerations

- All software ran on the same machine (no network latency)
- No software tuning of any kind was performed (e.g. no special JVM configuration was used)
- xAPI Statements contained only the elements required by the specification. Although considerably larger messages are possible the messages used in testing are representative of statements generated by Sakai.

Test Performed

Capacity Test

Test duration: 1 hour
Load type: ramp-up from 0 to 300 users adding a new thread every 10 seconds.

Each thread continuously generated between 60 and 360 statement POSTs and 1 statement GET for roughly each 100 statement POSTs. The openlrs elasticsearch index was dropped before starting this test (i.e. there was no openlrs data in elasticsearch).

Stress Test

Test duration: 10 min
Load type: ramp-up from 0 to 1000 users adding 2 new threads every second.

Each thread continuously generated between 60 and 360 statement POSTs and 1 statement GET for roughly each 100 statement POSTs. The openlrs elasticsearch index contained ~350000 records before starting this test.

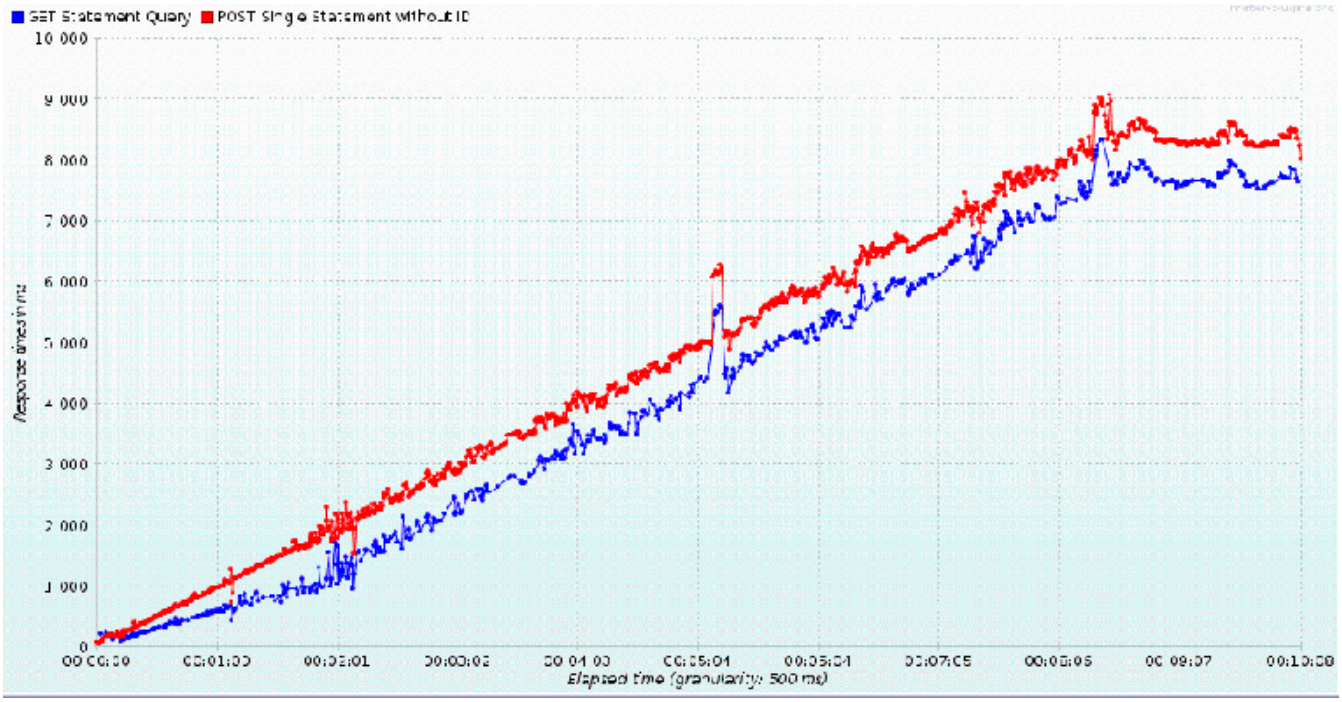
Test Results

Capacity Test

	Total Requests	Avg Response Time	Max Response Time	Error Rate	Request / Second
POST Statement	334554	188 ms	1964 ms	0%	
GET Statement	3389	106 ms	1289 ms	12%	
Totals	337943	188 ms	1964 ms	~.1%	92

Stress Test

	Total Requests	Avg Response Time	Max Response Time	Error Rate	Request / Second
POST Statement	71380	4726 ms	9099 ms	0	
GET Statement	679	4080 ms	7985 ms	15%	
Totals	72059	4719 ms	9099 ms	~.1%	120



Conclusion

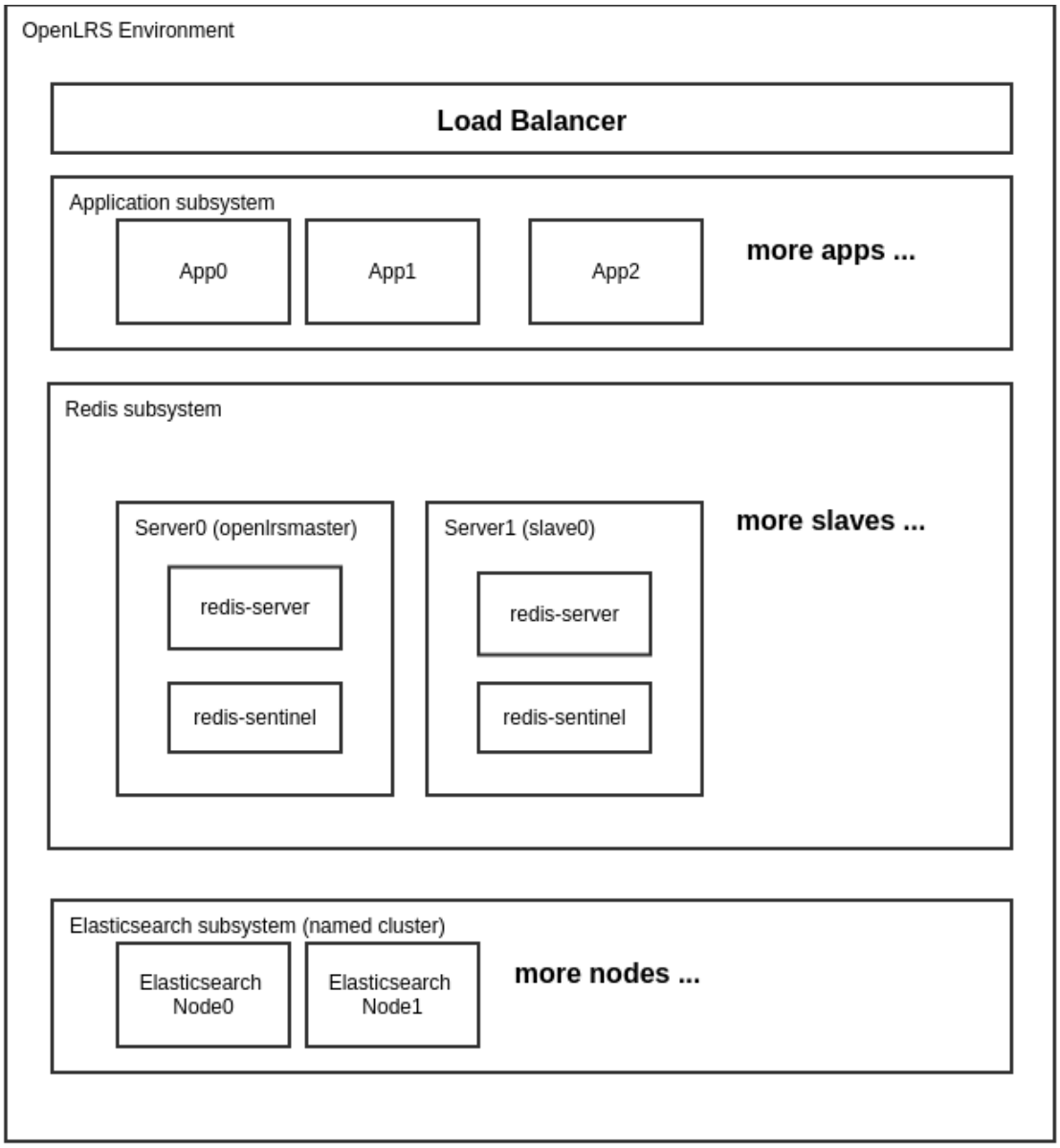
The capacity and stress tests show that OpenLRS performs reasonably well when considering the single application instance and lack of any tuning across the entire solution.

The capacity tests demonstrate that OpenLRS can handle sustained limited load (e.g. a 1000 student pilot). However, based on the stress test, there is work to be done to prior to a full scale institutional adoption. As most of the response time was spent waiting for connections (first to tomcat, second to elasticsearch) tuning will be the first course of action. Also of concern was the high error rate on GET Statement requests. The high error rate was, for the most part, a side effect of the eventual consistency strategy of queuing Statements to eventually be persisted. This strategy may be sufficient in many use cases however an alternative should be provided (likely at the cost of some performance) for use cases where Statements must be available in real time.

Overall the test goals were achieved. There is now a performance baseline that can be compared to and improved upon in subsequent versions of OpenLRS. Also, although OpenLRS does not require the use of Redis and Elasticsearch, these tests show that they were a good choice of tier one and tier two storage options and should be considered the de facto standard choices for OpenLRS going forward.

PRODUCTION ENVIRONMENT

The following recommendations are for a large-scale, highly available, production or production-like Open LRS environment.



Application servers

Installed components

- Java 7
- OpenLRS web application

Recommended directory structure

```
# OpenLRS Home
# - contains
# -- openlrs jar file
# -- openlrs start/stop script
# -- openlrs pid file (generated)
/export/home/openlrs

# OpenLRS Configuration
/export/home/openlrs/conf

# OpenLRS Logs
/export/home/openlrs/logs
```

Recommended Spring configuration

It is recommended that the configuration is stored in an external file (e.g. prod.properties) in the OpenLRS configuration directory.

```
# Active Profiles
# Default for a non-demo environment is redisElasticsearch
spring.profiles.active=redisElasticsearch

# Redis
# uncomment this line if you are using password protected redis
spring.redis.password=<somepassword>
# uncomment these lines if you are using redis sentinel for HA
spring.redis.sentinel.master=openlrsmaster
spring.redis.sentinel.nodes=<e.g. 127.0.0.1:26379,127.0.0.1:26380,127.0.0.1:26381>
# set this to half of your tomcat max threads
spring.redis.pool.max-idle=<1/2 tomcat max threads>
spring.redis.pool.min-idle=<set this to 10% of max-idle>
spring.redis.pool.max-active=<set this equal to max-idle>

# ELASTICSEARCH
spring.data.elasticsearch.cluster-name=<your cluster>
spring.data.elasticsearch.clusterNodes=<ip1:port1,ip2:port2...>

# LOGGING
logging.path=/export/home/openlrs/logs
logging.file=/export/home/openlrs/logs/openlrs.log
logging.config=classpath:logback.xml
logging.level.*=ERROR

# TOMCAT
# leaving max-threads at 0 will result in the default (200)
# recommended value 200xCPU (e.g. on AWS ec2 c3.large this value was 400)
server.tomcat.max-threads=0

# SECURITY
auth.basic.username=<some username>
auth.basic.password=<some password>
auth.oauth.key=<some key>
auth.oauth.secret=<some secret>

# INSTANCE
# Must be different for each instance in a clustered environment
instance.name=<some name>
```

Example JVM Options

```
-server -Xms2048m -Xmx2048m -XX:MaxPermSize=256m -XX:+UseParallelGC
-XX:MaxGCPauseMillis=1500 -XX:GCTimeRatio=9 -XX:+DisableExplicitGC
```

Sample start/stop script

This script could be used as is or converted easily to init.d.

```
#!/bin/sh
# ==================================
#
# A sample start/stop script for OpenLRS
#
```

```
# Update variables to suit your environment
#
# - JARFile: the OpenLRS application jar file
# - PIDFIle: the process id file created by the application must be named
openlrs.pid
# - SPRING_OPTS: Spring configuration options; typically this is just the path to a
properties file
# -- e.g.
SPRING_OPTS="-Dspring.config.location=/export/home/openlrs/prod.properties"
# - JAVA_OPTS: JVM options; Update to suit your environment
# -- e.g. JAVA_OPTS="-server -Xms2048m -Xmx4096m -XX:MaxPermSize=256m
-XX:+UseParallelGC -XX:MaxGCPauseMillis=1500 -XX:GCTimeRatio=9
-XX:+DisableExplicitGC"
#
# Example:
# JARFile="openlrs.jar"
# PIDFile="openlrs.pid"
# SPRING_OPTS="-Dspring.config.location=/export/home/openlrs/conf/prod.properties"
# JAVA_OPTS="-server -Xms2048m -Xmx2048m -XX:MaxPermSize=256m"
# JAVA_OPTS=$JAVA_OPTS" -XX:+UseParallelGC -XX:MaxGCPauseMillis=1500
-XX:GCTimeRatio=9 -XX:+DisableExplicitGC"
#
# ==================================

JARFile=""
PIDFile=""
SPRING_OPTS=""
JAVA_OPTS=""

check_if_pid_file_exists() {
    if [ ! -f $PIDFile ]
    then
        echo "PID file not found: $PIDFile"
        exit 1
    fi
}
check_if_process_is_running() {
    if ps -p $(print_process) > /dev/null
    then
        return 0
    else
        return 1
    fi
}
print_process() {
    echo `cat $PIDFile`
}
case "$1" in
status)
    check_if_pid_file_exists
    if check_if_process_is_running
    then
        echo $(print_process)" is running"
    else
        echo "Process not running: $(print_process)"
    fi
    ;;
stop)
    check_if_pid_file_exists
```

```
    if ! check_if_process_is_running
    then
        echo "Process $(print_process) already stopped"
        exit 0
    fi
    kill -9 $(print_process)
    rm -f $PIDFile
    echo "Process stopped"
    ;;
start)
    if [ -f $PIDFile ] && check_if_process_is_running
    then
        echo "Process $(print_process) already running"
        exit 1
    fi
    nohup java $JAVA_OPTS $SPRING_OPTS -jar $JARFile > /dev/null 2>&1 &
    echo "Process started"
    ;;
*)
    echo "Usage: $0 {start|stop|status}"
```

```
    exit 1
esac
exit 0
```

Redis servers

Installed components

- Redis 2.8+

Recommended installation

Install redis from a repository (apt-get install redis-server or yum) or follow the manual instructions below.

This is adapted from redis installation guide see the 'Installing Redis more properly' section.

Manual installation

```
# Assumes:
# - executables redis-server, redis-cli, redis-sentinel have been copied to
/usr/local/bin

# Make conf directory
sudo mkdir /etc/redis

# Make data directory
sudo mkdir /var/redis
sudo mkdir /var/redis/6379

# Copy the init script for redis-server
# This script can be found in the utils directory of the redis distribution or see
redis-server init script below
# NOTE if you are running redis on a port other than 6379 update REDIS_PORT in the
init script and throughout these instructions to match your port
sudo cp utils/redis_init_script /etc/init.d/redis_6379

# Copy the base redis configuration
# This file (redis.conf) can be found in the root of the redis distribution
sudo cp redis.conf /etc/redis/
# Copy the local configuration overrides
# see redis-master.conf and redis-slave.conf below
# NOTE update properties to suit your environment
sudo cp redis-<type>.conf /etc/redis/6379.conf

##
# If using redis sentinel for high availability

# Copy the init script for redis-sentinel
# This script can be found below see
# NOTE if you are running redis on a port other than 26379 update REDISSENTINEL_PORT
in the init script and throughout these instructions to match your port
sudo cp <script> /etc/init.d/redissentinel_26379

# Copy the sentinel configuration
# This file can be found below - sentinel.conf
sudo cp sentinel.conf /etc/redis/26379.conf
##

# Add the init scripts
sudo update-rc.d redis_6379 defaults
sudo update-rc.d redissentinel_26379 defaults

# Start redis server
/etc/init.d/redis_6379 start

# Start redis sentinel
/etc/init.d/redissentinel_26379 start
```

Recommended conf

redis-master.conf

```
include /etc/redis/redis.conf

daemonize yes
pidfile /var/run/redis_6379.pid
logfile /var/log/redis_6379.log
dir /var/redis/6379

# password for the redis master node
requirepass <somepassword>
# the public ip for the server the redis master node is running on
bind <masterpublicip>


# Allocate more memory for slaves and pubsub
# What this is saying is if the output buffer (i.e. the amount of data waiting to be
written to a slave)
# reaches 256mb total or is 128mb or more for 60 seconds stop accepting data
# NOTE adjust these values to fit the memory constraints of your environment
# NOTE The Redis memory usage calculation does not take the replication buffer size
into account
client-output-buffer-limit slave 256mb 128mb 60
client-output-buffer-limit pubsub 64mb 32mb 60
# If the above output buffers fill up, don't accept any more writes
maxmemory-policy noeviction
```

redis-slave.conf

```
include /etc/conf/redis.conf

daemonize yes
pidfile /var/run/redis_6379.pid
logfile /var/log/redis_6379.log
dir /var/redis/6379

# password for this redis slave node must match master password if using sentinel
requirepass <somepassword>
# the public ip for the server this redis slave node is running on
bind <slavepublicip>
# master redis node
slaveof <masterpublicip> <masterport>
# password for master node
masterauth <somepassword>

# Allocate more memory for slaves and pubsub
# What this is saying is if the output buffer (i.e. the amount of data waiting to be
written to a slave)
# reaches 256mb total or is 128mb or more for 60 seconds stop accepting data
# NOTE adjust these values to fit the memory constraints of your environment
# NOTE The Redis memory usage calculation does not take the replication buffer size
into account
client-output-buffer-limit slave 256mb 128mb 60
client-output-buffer-limit pubsub 64mb 32mb 60
# If the above output buffers fill up, don't accept any more writes
maxmemory-policy noeviction
```

sentinel.conf

```
# sentinel monitor <master-name> <ip> <redis-port> <quorum>
# quorum value depends on how many sentinels you are running
# e.g. 3 sentinels quorum should be 2, 2 sentinels quorum should be 1
sentinel monitor openlrsmaster <masterpublicip> <masterport> <quorum>
# openlrsmaster is marked down after 30 seconds of ping failures
sentinel down-after-milliseconds openlrsmaster 30000
# a new master is selected after one minute
sentinel failover-timeout openlrsmaster 60000
sentinel parallel-syncs openlrsmaster 1
sentinel auth-pass openlrsmaster <somepassword>
```

redis-server init script

```
#!/bin/sh
#
# Simple Redis init.d script conceived to work on Linux systems
# as it does use of the /proc filesystem.
# NOTE add these three comments if running on centos
# chkconfig:   - 85 15
# description:  Redis is a persistent key-value database
# processname: redis_6379
REDISPORT=6379
EXEC=/usr/local/bin/redis-server
CLIEXEC=/usr/local/bin/redis-cli
PIDFILE=/var/run/redis_${REDISPORT}.pid
CONF="/etc/redis/${REDISPORT}.conf"
case "$1" in
    start)
        if [ -f $PIDFILE ]
        then
                echo "$PIDFILE exists, process is already running or crashed"
        else
                echo "Starting Redis server..."
                $EXEC $CONF
        fi
        ;;
    stop)
        if [ ! -f $PIDFILE ]
        then
                echo "$PIDFILE does not exist, process is not running"
        else
                PID=$(cat $PIDFILE)
                echo "Stopping ..."
                $CLIEXEC -p $REDISPORT shutdown
                while [ -x /proc/${PID} ]
                do
                    echo "Waiting for Redis to shutdown ..."
                    sleep 1
                done
                echo "Redis stopped"
        fi
        ;;
    *)
        echo "Please use start or stop as first argument"
        ;;
esac
```

redis-sentinel init script

```
#!/bin/sh
#
# Simple Redis Sentinel init.d script conceived to work on Linux systems
# as it does use of the /proc filesystem.
# NOTE add these three comments if running on centos
# chkconfig:   - 85 15
# description:  Redis is a persistent key-value database
# processname: redissentinel_26379
REDISSENTINELPORT=26379
EXEC=/usr/local/bin/redis-sentinel
CLIEXEC=/usr/local/bin/redis-cli
PIDFILE=/var/run/redissentinel_${REDISSENTINELPORT}.pid
CONF="/etc/redis/${REDISSENTINELPORT}.conf"
case "$1" in
    start)
        if [ -f $PIDFILE ]
        then
                echo "$PIDFILE exists, process is already running or crashed"
        else
                echo "Starting Redis sentinel..."
                $EXEC $CONF
        fi
        ;;
    stop)
        if [ ! -f $PIDFILE ]
        then
                echo "$PIDFILE does not exist, process is not running"
        else
                PID=$(cat $PIDFILE)
                echo "Stopping ..."
                $CLIEXEC -p $REDISSENTINELPORT shutdown
                while [ -x /proc/${PID} ]
                do
                    echo "Waiting for Redis sentinel to shutdown ..."
                    sleep 1
                done
                echo "Redis sentinel stopped"
        fi
        ;;
    *)
        echo "Please use start or stop as first argument"
        ;;
esac
```

Elasticsearch servers

Installed components

- Java 7 (Elasticsearch recommends Oracle JVM)
- Elasticsearch 1.3+

Installation

Download or install via repository

System configuration

```
# Ensure elasticsearch has the ability to lock memory
sudo ulimit -l unlimited
```

elasticsearch configuration

The elasticsearch configuration file (elasticsearch.yml) can be found in /etc/elasticsearch if you installed elasticsearch as a service (recommended) otherwise it is found in $ES_HOME/config. At a minimum you will want to adjust the following properties

```
cluster.name: <your cluster>

node.name: <your node name>

# where to store data
path.data: /path/to/data
# where to store logs
path.logs: /path/to/logs

bootstrap.mlockall: true

network.host: <your ip>


# If running in a cluster (recommended)
# If running your cluster on a local network
# then you can use the default dynamic multicast feature to find cluster members

# If ec2 cluster see http://www.elasticsearch.org/tutorials/elasticsearch-on-ec2/

# Otherwise if you know the cluster members
# disable multicast
discovery.zen.ping.multicast.enabled: false
# list each cluster member
discovery.zen.ping.unicast.hosts: ["host1:port"]

indices.store.throttle.type: none
threadpool.index.queue_size: 400
```

Update memory configuration in /etc/default/elasticsearch (assumes running as service)

```
ES_HEAP_SIZE=<as much memory as you can spare>
```

After these updates are complete run

```
sudo update-rc.d elasticsearch defaults 95 10
sudo /etc/init.d/elasticsearch start
```

PRODUCTION PERFORMANCE

Summary

The following are the results of performance tests for OpenLRS in a production-like environment (as documented in the section above). Performance tests were executed using JMeter test scripts developed by the Apereo Learning Analytics community (https://github.com/Apereo-Learning-Analytics-Initiative/LRSLoadTest).

Overall OpenLRS performed well. OpenLRS can easily handle 250-300 requests per second or about a million requests per hour.

Test Objectives

- Determine performance characteristics of a production-like deployment of OpenLRS
- Validate software design and architecture choices
- Tune the application as necessary

Test Conditions

Software

- Amazon Linux (Centos)
- OpenLRS (TwoTierRepository configuration with embedded Tomcat 7)
- Redis 2.8.13
- Elasticsearch 1.3.2
- OpenJDK 1.7.0_65

Hardware (AWS ec2 c3.large)

- vCPU: 2
- Memory: 3.75 GiB
- Processor: High Frequency Intel Xeon E5-2680 v2 (Ivy Bridge)
- OS-type: 64 bit
- SSD Storage: 2x16 GB

Other considerations

- Test were executed in AWS VPC

Test Performed

Capacity Test 1

Test duration: 1 hour
Load type: ramp-up from 0 to 300 thread adding a new thread every 10 seconds.

Each thread continuously generated between 60 and 360 statement POSTs and 1 statement GET for roughly each 1000 statement POSTs. The openlrs elasticsearch index was dropped before starting this test (i.e. there was no openlrs data in elasticsearch).

Capacity Test 2

Test duration: 8 hours
Load type: ramp-up from 0 to 300 thread adding a new thread every 5 seconds.

Each thread continuously generated between 30 and 60 statement POSTs and 1 statement GET for roughly each 1000 statement POSTs. Th e openlrs elasticsearch index contained approximately ten thousand records before starting this test.

Stress Test 1

Test duration: 10 min
Load type: ramp-up from 0 to 1000 threads adding 2 new threads every second.

Each thread continuously generated between 60 and 360 statement POSTs and 1 statement GET for roughly each 1000 statement POSTs. The openlrs elasticsearch index contained approximately two million records before starting this test.

Stress Test 2

Test duration: 10 min

Load type: ramp-up from 0 to 2500 threads adding 8 new threads every second.

Each thread continuously generated between 500 and 1000 statement POSTs and 1 statement GET for roughly each 1000 statement POSTs. The openlrs elasticsearch index contained approximately nine million records before starting this test.

Failover Test 1 (app instance)

Test duration: 10 min
Load type: ramp-up from 0 to 100 thread adding a new thread every 3 seconds.

Each thread generated between 99 and 101 statement POSTs for roughly 10000 requests during the test. One of three app instances was taken offline two minutes into the test.

Failover Test 2 (redis node)

Test duration: 10 min
Load type: ramp-up from 0 to 100 thread adding a new thread every 3 seconds.

Each thread generated between 99 and 101 statement POSTs for roughly 10000 requests during the test. One of two redis instances was taken offline two minutes into the test.

Failover Test 3 (elasticsearch node)

Test duration: 10 min
Load type: ramp-up from 0 to 100 thread adding a new thread every 3 seconds.

Each thread generated between 99 and 101 statement POSTs for roughly 10000 requests during the test. One of two elasticsearch instances was taken offline two minutes into the test.

Test Results

Capacity Test 1

	Total Requests	Avg Response Time	Max Response Time	Error Rate	Request / Second
POST Statement	1016506	137 ms	5510 ms	0%	282
GET Statement	1032	125 ms	1225 ms	97%	
Totals	1017538	137 ms	5510 ms	~.1%	282

Capacity Test 2

	Total Requests	Avg Response Time	Max Response Time	Error Rate	Request / Second
POST Statement	8435469	167 ms	5862	0%	293
GET Statement	8394	146 ms	1856	95%	
Totals	8443863	167 ms	5862	.1%	293

Stress Test 1

	Total Requests	Avg Response Time	Max Response Time	Error Rate	Request / Second
POST Statement	171221	252 ms	7788 ms	0%	283
GET Statement	158	206 ms	2140 ms	85%	
Totals	171379	252 ms	7788 ms	.08%	283

Stress Test 2

	Total Requests	Avg Response Time	Max Response Time	Error Rate	Request / Second
POST Statement	153541	231 ms	7301 ms	.04%	251
GET Statement	172	238 ms	4777 ms	21%	
Totals	153713	231 ms	7301 ms	.06%	251

Failover Test 1 (app instance)

Total Requests	Failed Requests	Data loss
10001	8	.0007%

Failover Test 2 (redis node)

Total Requests	Failed Requests	Data loss
10001	1037	12%

Failover Test 2 (elasticsearch node)

Total Requests	Failed Requests	Data loss
10003	0	0%

Conclusion

The capacity and stress tests show that OpenLRS performs well and is capable of handling production scale. Failover tests demonstrate that OpenLRS can provide the high availability needed for production.

The capacity tests demonstrate that OpenLRS can handle sustained load. In each capacity test roughly one million records were indexed per hour. It is hard to say with certainty how many records per hour will be produced by a typical Sakai installation however one million seems like a reasonable number (~1500 users producing 10 xAPI statements per minute). Pilot courses will help to refine these estimates and will also help with capacity planning especially in the area of data storage. One million Sakai-like xAPI statements indexed in Elasticsearch uses approximately .7 gigabytes of disk space - at this level of usage disk space will go fast so planning and migration strategies will be key.

Stress tests showed significant improvement over the baseline tests. Largely this was due to tuning of Tomcat and Redis connection pools and the presence of additional application instances.

Failover tests were very promising. Application failover resulted in minimal request failures and, more importantly, very little data loss. Elasticsearch failover resulted in no request failures (as expected) and no data loss. Only Redis failover had a significant impact on the system with a high percentage of request failures and thus significant data loss. The main issue here is because Redis uses a master-slave setup for high availability there will always be some period of time after the failure of the master node where Redis is unavailable (during this period the slave nodes are determining a new master). For these tests the selection interval was one minute which correlates to the number of failed requests. This value is configurable (see failover-timeout in sentinel.conf).

Hardware Recommendations

Application instances (linux)

- key considerations: cpu and memory
  - CPU: minimum 2 cores (additional cores allow improved concurrency)
  - Memory: 4-8 gb
  - Disk: 16 gb (minimal storage - logging only)
- required supporting software:
  - JDK/JRE 1.7.x

Redis instances (linux)

- key considerations: none (general purpose instances such as ec2 m3 types are sufficient)
    - CPU: single core (favor clock speed)
    - Memory: 4 gb
    - Disk: 16 gb (minimal storage - logging only)
- required supporting software:

Elasticsearch instances (linux)

- key considerations: memory and disk
    - CPU: minimum 2 cores (favor cores over clock speed)
    - Memory: 16 gb minimum / 64 gb max
    - Disk: 100 gb minimum (depends on transaction volume) SSD required
- required supporting software:
    - JDK/JRE 1.7.x