**Programming Assignment 3**

Asterix and the Olympic Games (Fault Tolerant edition)

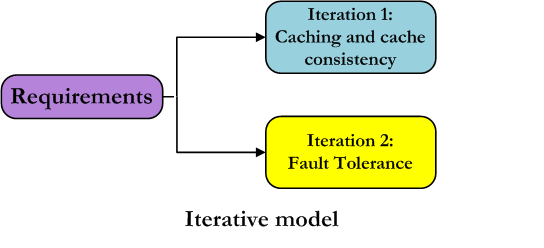
**Ashish Jain**

**Sinthu Muralitharen**

**Section 1. Introduction**

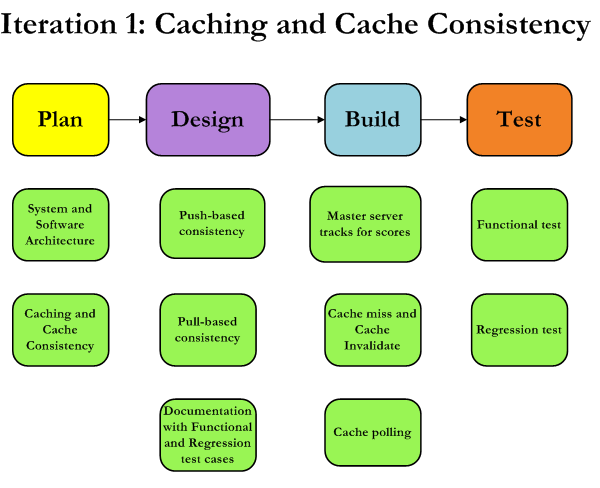
Asterix and Olympics game (Fault Tolerant Edition**)**, in this release the performance and reliability enhancements are implemented. To enhance performance, a **cache** is added to each front end server. When there is a score request, the front-end servers first check the cache to see if the requested data is cached. Also, as the scores are frequently updated, cache consistency is much needed. To ensure reliability, the crash failures of the front-end servers are handled. The system is designed for the efficient score updates for the clients based on the conditions in the problem statement.

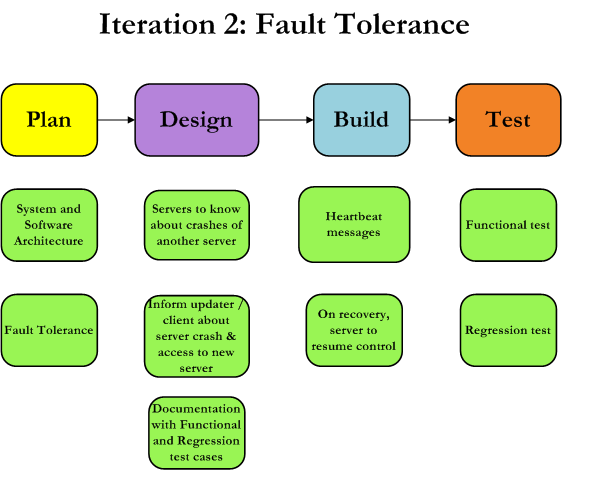
**Section 2. Design**

****

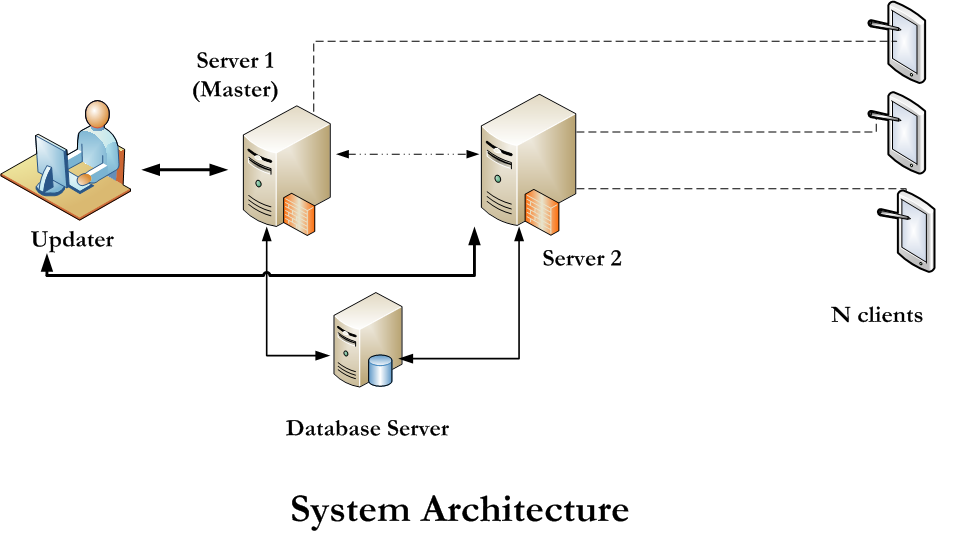
We used the Iterative model of the SDLC to design the system. Iterative process starts with a simple implementation of a small part of the software requirements and iteratively enhances the future versions until the full system is implemented. At each iteration, design modifications are made and new functionalities are added. The system is developed through repeated cycles and in smaller portions at a time.

In the Iteration 1, we have implemented the first part of the project requirement, i.e caching and cache consistency. In the iteration 2, we implemented the fault tolerance of the system. The figures below explain the detailed phases of the build.

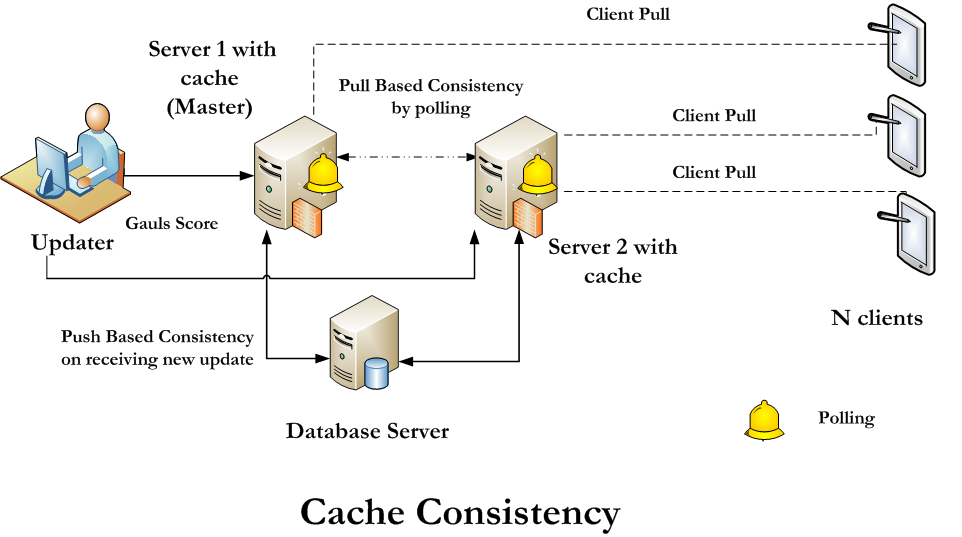




**Section 3. System Architecture**

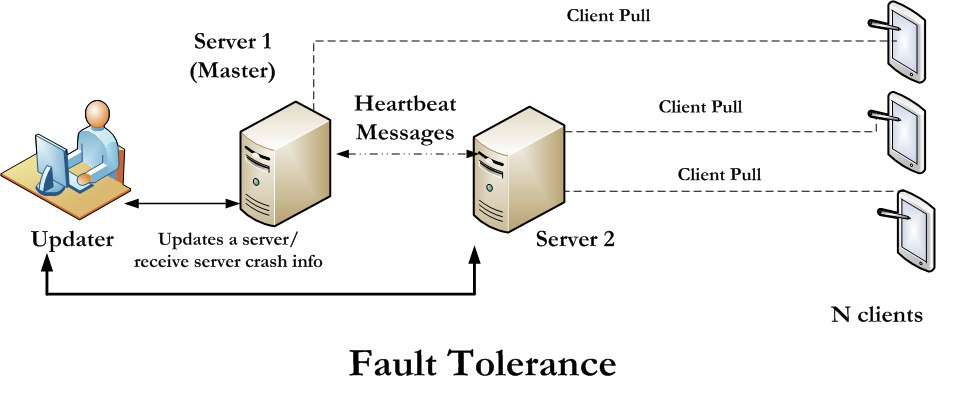
****

**Section 4. Software Architecture**

****

**To enhance performance**, a cache is added to each front end server. The cache stores a copy of recently accessed scores for each sport. When client requests score to the server, the front-end server first checks the cache to see if the requested data is cached. In the event of a cache hit, the cached data is used to service the request. If the request results in a cache miss, the front-end servers request data from the backend server (database). Since the scores are frequently updated, cache consistency is a needed. Two approaches to cache consistency are implemented: push-based consistency and pull-based consistency.

In **push-based consistency**, the master front-end server tracks the scores that are cached within each cache and upon receiving an update, send a cache invalidate to any cache that caches that data. Cache invalidates message cause the cache to remove the corresponding item from the cache and a subsequent request causes a cache miss and the item is brought back into the cache.

In **pull-based consistency**, the cache to periodically polls the front-end server to check if the data has changed. If data is unchanged, it remains in the cache, otherwise the old data is removed. It has to access database every time to check whether data is stale or not in the cache.

**To ensure reliability (Fault Tolerance)**, the front-end servers are designed to handle crash failures. Since the servers are stateful and store the client state to recover, we assume that only one of the front-end servers can fail at any point. Heartbeat messages are implemented to detect the presence of a failure and have the other front-end server to take over the tasks of the failed server. When the failed server recovers, it resumes its control and whole system is back to default operation. This implementation is extensible for any number of clients and server. The server with highest available ID is selected as the master server to take over the responsibility of failed server.

**Section 5.1. Functional Test Cases**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.No** | **Test Scenarios** | **Priority** | **Input** | **Expected Output** |
| **1** | Periodic score update from to client  from cache | 1 | NA | Score to be fetched from the local cache and sent to client |
| **2** | Fresh score request and not in cache (cache-miss) | 1 | NA | On cache miss score to be fetched from database and sent to client |
| **3** | New score from cacophonix to server | 1 | NA | 1. Old score in cache to be invalidated and sent to database  2. New score to be cached. |
| **4** | Cache polling | 1 | NA | Cache to check front end server for updated score |
| **5** | Server 1(Master) crash | 1 | NA | Heartbeat message to communicate other server to handle all requests in the system. |
| **6** | Server 2 crash | 1 | NA | Heartbeat message to communicate Master server to handle all requests in the system. |
| **7** | Server 1 (Master) recovered | 1 | NA | To resume its control and restore to default system operations |
| **8** | Server 2 recovered | 1 | NA | To resume its control and restore to default system operations. |
| **9** | Database not started | 2 | NA | In case database is down, server should display warning on receiving client request. |
| **10** | If both front end servers are down | 2 | N/A | Clients should detect when both the servers got crashed. |
| **11** | If client randomly select some server to connect to which is not up | 2 | N/A | Client will find alternate server to register rather than not starting |
| **12** | Client doesn’t exist | 2 | N/A | In case client has been killed or crashed, server should not be affected by it. |
| **13** | Both front end servers should use either of the cache algorithm | 2 | N/A | In case they are using different policy, system will crash eventually |

**Section 5.2. Regression Test Cases**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.No** | **Test Scenarios** | **Priority** | **Input** | **Expected Output** |
| **1** | Presence of update window | 1 | - | Update window to be opened |
| **1.1** | Enter options to update 1 to update score or 2 to update medal tally | 1 | 1 to update score 2 to update medal tally | Next question to be triggered |
|  |  |  | Any other input | Error message to be popped |
| **1.3** | Enter team name as 'Gauls' or 'Rome' | 1 | 'Gauls' or 'Rome' in exact format | Next Question to be triggered |
|  |  |  | Any other value than specified format | Error message to be popped |
| **1.4** | Enter the event type | 1 | Event type in exact format | Next Question to be triggered |
|  |  |  | Any other value than specified format | Error message to be popped |
| **1.5** | Enter the Score | 2 | Valid Number | 1) An update in the server,  DONE UPDATE  Should be displayed  2) Periodically updated to unregistered clients and registered clients. |
|  |  |  | Any other value than a valid number format | Error message to be popped |
| **1.6** | Enter options to update 1 to update score or 2 to update medal tally | 1 | 1 to update score 2 to update medal tally | Next question to be triggered |
|  |  |  | Any other input | Error message to be popped |
| **1.7** | Enter team name as 'Gauls' or 'Rome' | 1 | 'Gauls' or 'Rome' in exact format | Next Question to be triggered |
|  |  |  | Any other value than specified format | Error message to be popped |
| **1.8** | Enter the event type | 1 | Event type in exact format | Next Question to be triggered |
|  |  |  | Any other value than specified format | Error message to be popped |
| **1.9** | Enter the medal type | 1 | Event medal type in exact format | Next Question to be triggered |
|  |  |  | Any other value than specified format | Error message to be popped |
| **2** | Presence of Server 1 | 1 |  | Server 1 is ready |
| **3** | Presence of Client | 1 |  | Client Process is running status is displayed for every client created |
| 4 | Enter options to update 1 to update score or 2 to update medal tally | 1 | 1 to update score 2 to update medal tally | Score to be updated with timestamp.  Next question to be triggered |
|  |  |  | Any other input | Error message to be popped |
| 4.2 | Enter the Score | 2 | Valid Number | 1) An update in the Data Base server, DONE UPDATE  Should be displayed  2) Periodically time stamped score updated to the clients. |
|  |  |  | Any other value than a valid number format | Error message to be popped |
| 4.3 | Enter options to update 1 to update score or 2 to update medal tally | 1 | 1 to update score 2 to update medal tally | Score to be updated with timestamp  Next question to be triggered |
|  |  |  | Any other input | Error message to be popped |
| 5 | Presence of Server 2 | 1 |  | Server 2 is ready |
| 6 | Presence of Database server | 1 |  | Database server is ready |

**Section6. Test Case Designs:**

1. **Fault Tolerance & Recovery:** In this test case, we will handle following aspects of fault tolerance:

* How the heartbeat message between two front-end servers handle the failure of either of the servers.
* It also demonstrates the transfer of responsibility of failed server to LIVE server. Therefore clients registered with failed server will now pull score from LIVE server and cacophonix will send updates to LIVE server.
* Recoverability is also implemented. Therefore, whenever the previously crashed server starts again, it will get back its original task to perform.

**Check this test case as follows:**

* **Run nameserver:** python nameserver.py. (**Note:** In case this command throws error, move to next step because nameserver is already running.)
* **Run server1:** python server1.py <argument-(1/2)>.This will start with master server with ID=1. It receives update from cacophonix.

**NOTE: argument –**Enter the argument as integer either 1 or 2. 1 will run push based cache consistency and 2 will run pull based cache consistency.

* **Run server2:** python server2.py <argument-(1/2)>**.** This is the second front-end server with ID=2.

**NOTE: argument –**Enter the argument as integer either 1 or 2. 1 will run push based cache consistency and 2 will run pull based cache consistency.

* **Run client:** python client.py**.** It will randomly register the client to either of the front-end server. You can run multiple instances of client.py.
* Now **kill** either of server1 or server2 by running **./killServer1.sh or ./killServer2.sh**. When one of the servers is killed, other server will detect it and take over its responsibility.

**WARNING:** kill scripts should be on same elnux machine from which server scripts are run. Otherwise use **‘ps –x’** command to find PID of server process and kill it using **“kill -9 <PID>”.**

* Now start back the killed server. It will recover to its original state and start performing the original tasks.

1. **Push Based Cache Consistency:** In push based cache consistency, the master front-end server when receives the update from Cacophonix, will send invalidate signal to caches storing that data. In this way stale data is removed from cache. When server receives request corresponding to that data, the cache miss will occur and it will access database to retrieve data. In this step cache is also populated with new or current score/medal.

To test this, run **following steps**:

* **Run nameserver:** python nameserver.py. (**Note:** In case this command throws error, move to next step because nameserver is already running.)
* **Run server1:** python server1.py 1 (**Note**: argument *‘1’* means, it is push based caching)
* **Run server2:** python server2.py 1

**NOTE:** Both the server should have same argument 1. We can’t have different cache consistency on different servers.

* **Run database server:** python database.py
* **Run clients**: python client.py (**Note:** Any number of clients can be run).
* **Run update server:** python updateserver.py

After running the above setup, you will see appropriate messages on screen, where on each update, corresponding cache get invalidated. On request from client, the cache gets populated.

1. **Pull Based Cache Consistency:** In pull based cache consistency, the cache will probe the front-end server for updated scores in every 5seconds. Front-end server has to access database in order to check for the updated score. If stale data is found in the cache, it will immediately be removed. On subsequent requests from client, updated data will be brought back into the cache.

To test this, run **following steps**:

* **Run nameserver:** python nameserver.py. (**Note:** In case this command throws error, move to next step because nameserver is already running.)
* **Run server1:** python server1.py 2 (**Note**: argument *‘2’* means, it is pull based caching)
* **Run server2:** python server2.py 2

**NOTE:** Both the server should have same argument ‘*2’*. We can’t have different cache consistency on different servers.

* **Run database server:** python database.py
* **Run clients**: python client.py (**Note:** Any number of clients can be registered. Clients will get randomly registered to any of the server1 or server2).
* **Run update server:** python updateserver.py

After running the above setup, we will see messages displaying the polling of cache in every 5 seconds. If stale data is found, the data will be removed and appropriate message is displayed on server screen with which client is registered).

**Section7. How to Run System**

Same Instructions can be found in Readme.txt with in lab3 directory containing code:

* **Run nameserver**: python nameserver.py (**Note:** In case this command throws error, move to next step because nameserver is already running.)
* **Run database server**: python databaseserver.py
* **Run server 1**: python server1.py <argument> (dicts.py and lock.py should be in same folder as this script file)
* **Run server 2**: python server2.py <argument>

**NOTE**: **argument –** Enter the argument as integer either *‘1’ or ‘2’*. 1 will run push based cache consistency and 2 will run pull based cache consistency.

* **Run clients**: python client.py (**Note:** Any number of clients can be run).
* **Run Updateserver:** python updateserver.py. It will read input from input.txt file.
* **Note1:** In order to **kill server1 or server2**, use ./killServer1.sh or ./killServer2.sh. To kill client or any other process, please use ‘ps –x’ to check process PID and use ‘kill -9 <PID>.

**WARNING:** kill scripts should be on same elnux machine from which server scripts are run. Otherwise use **‘ps –x’** command to find PID of server process and kill it using **“kill -9 <PID>”.**

**Note2:** database file will be generated with name db.p. In this file, data is stored in pickle format, hence it is not human readable. We can load it into a program and use it.

**Note3:** Appropriate **informative messages** will be printed whenever a server crashes,client is registered to other server, cache is missed or item is found in cache etc.

**Section8.** **Design Tradeoff considered and made**

* In our approach, we can run any number of front-end server and clients. Each server is assigned unique ID number. When any server fails, the server with highest available ID is take over the failed server’s responsibility.
* All the front-end servers exchange heartbeat messages amongst each other which will help them in finding out if any server has crashed in the distributed system.
* In case client randomly selects either of server 1 or 2 to get registered with. If that particular server is down or has crashed, client will search for alternate server to get registered with.
* We are implementing caching mechanism such that entire system either uses push based or pull based cache consistency. In case of pull based policy, predefined polling interval has been used rather than using popularity measure to decide interval.
* In cache consistency approaches, while polling the front-end server, if stale data is found in cache, it is removed from the cache. In this approaches, we are not updating the cache immediately with new data, rather we wait for subsequent client request to bring data into cache. This leads to overhead because we are already accessing database to check for data consistency but we are not bringing in new data in cache.

**Section 9: Future Improvements and Extensions**

* Instead of just removing stale data from cache while polling, we can update cache as well.
* Front-end servers should be made stateless. Therefore, even if entire system crashes down, it can recover from the crash upon starting.
* In future improvement, we would like to integrate database server to a MySQL database to store data instead of using pickle file.
* Currently, both the server either uses pull based or push based cache consistency mechanism. In future, we can implement cache consistency is such a way that server can individually decide upon using caching mechanism.

**Section 10: Performance Results**

* We are using pickle serializer to pass data over the network. Latency is very less because pickle is known for fast data transmission. However, it has security vulnerability but since we are not dealing with security threats as of now, pickle seems as a reasonable option. We can change it to other serializer option in future if security requirement comes up.
* **Latency of Client pulling the scores (Average Response Time):** Latency for client pulling the score varies each time the experiment is conducted with different number of clients. Sometimes even on increasing the number of client, the latency decreased. This can be widely attributed to load on the edlab server. But the time taken for serving one pull request is on average close to: **0.02 seconds**. However, average is lower than the average time reported in previous lab, which is **0.05 seconds**. Hence, addition of caches to front-end servers reduces the overall time to serve client request.
* **Push based** cache consistency performs better than **pull based** here. Since the scores are updated frequently, the push approach keeps the cache updated, while pull based techniques often lead client receiving stale score. The stale data in pull based consistency depends on the polling interval. Since we have kept the interval to be small i.e. 5seconds, the stale data seen in cache is quite low.
* To be precise, we didn’t get much noticeable difference in performance by increasing the clients. However, we have not run 20-50 clients, so that we have to do in future steps.

**Section 11. Output**

**Master Server:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

MSG: Updating medal\_cache..

[u'0', u'0', u'0']

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

MSG: Checking medals in cache..

NOTE: Cache missed..

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

MSG: Updating medal\_cache..

[u'0', u'0', u'0']

Server 2 is alive..

Server 2 is alive..

Set the score for database

the score is set

MSG: Invalidating the cache...

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

MSG: Checking scores in cache..

NOTE: Cache missed..

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

MSG: Updating score\_cache..

[u'20', u'30']

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

MSG: Checking scores in cache..

found in cache

**Output from Client: Prints from which server output has been received and the score/medal.**

Registered with Server: 1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Score received from Server: 1

Score: snowboard - Gauls:0 Romans:0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Medal Tally received from Server: 1

Medal: Gauls gold:0 silver:0 bronze:0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Medal Tally received from Server: 1

Medal: Romans gold:0 silver:0 bronze:0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Server sending the score is Server:1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Score received from Server: 1

Score: skating - Gauls:20 Romans:30

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Score received from Server: 1

Score: curling - Gauls:0 Romans:0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Score received from Server: 1

Score: snowboard - Gauls:40 Romans:50

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Medal Tally received from Server: 1

Medal: Gauls gold:1 silver:0 bronze:0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Medal Tally received from Server: 1

Medal: Romans gold:0 silver:0 bronze:0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*