Warsaw University of Technology

Faculty of Electronics and Information Technology

Dimitrios Georgousis, K-7729

Gabriel Paic

Distributed Computing and Systems

Project Documentation

Introduction

Topic: 13 write – update. Implement distributed memory with read and write access, using write-update protocol.

We present a concept design for the above distributed system which reflects our current vision of implementation. The aim of the project is educational.

Assumptions

This distributed system is a developer/programmer tool. Offering an interface to use distributed memory. It uses the main memory of its Nodes and doesn't utilize their filesystems for saving data (data is not persistent). The system is in essence a DSM simulator with a very simple cache implementation. Focus was put on showcasing the write-update and communication protocols.

It should be pointed out that from now on 'Server' and 'Node' are going to be used to refer to the same entities in our Distributed Shared Memory System.

Use Cases

name	description	return value (JSON format)
Use Cases		
connect()	Connects to the server with a specific <server_address>	-
disconnect()	Disconnects from the server that the user is already connected to	{ “status”: num, “message”: str }
read()	Reads from a specific <memory_address> in the system	{ “status”: num, “message”: str, “data”: obj, “istatus”: str, “wtag”: num, “ltag”: num }
write()	Writes to a specific <memory_address> in the system	{“status”: num, “message”: str }
Auxiliary Use Cases (used for testing/checking errors)		
lock()	Performs an acquire_lock() operation on the lock protecting a specific <memory_address>	{“status”: num, “message”: str, “ret_val”: boolean, “ltag”: str, “wtag”: str }
unlock()	Performs a release_lock() operation on the lock protecting a specific <memory_address>	{“status”: num, “message”: str, “ret_val”: boolean, “ltag”: str, “wtag”: str }
dumpcache()	Returns to the user the cache contes of the Server they are connected to	{“status”: num, “message”: str, “cache”: list }

Let’s explain the above fields:

- “status”: is 0 if the operation succeeded, non-zero otherwise
- “message”: is a short descriptive message of the operation, useful for logging events that happened in the system
- “wtag”: is the last write tag, a timestamp of the last time something was written to that address
- “ltag”: is the last lock tag, a timestamp of the last time that lock was used
- “data”: our implementation of the DSM, since it is high-level, allows any object type to be stored in the memory addresses. Effectively, we can say that each memory address can act as a whole page depending on how large the data the user stores in them is
- “istatus”: status of the data item in a specific <memory_address>. It is either “E”: exclusively owned by its host server, or “S”: shared between the host and other Servers
- “ret_val”: indicates if our operation managed to lock (or unlock) a given lock
- “cache”: a list of cached items on a specific Server

The system has one user type that is able to perform all operations on it. The above interfaces are provided as simple programs that take input/give output to the console. Also, we have written an abstraction which allows one to use clients in either language through it. This abstraction is provided as a Python module which can be imported and used in other code.

System - Architecture

The languages used in our implementation are Python and Java. The operating systems are Windows 11 Pro and Ubuntu 22.04.4 LTS through WSL. The Linux machine runs a Python and a Java Server. The Windows machine runs a Java Server and a Python client (or Java client, there is not much difference).

We deploy 3 Servers (Nodes) intending to showcase the functionality of the copy holder chain of memory items (data corresponding to a specific <memory_address> in the system). If we had only 2 Servers then the copy holder chain would be trivial. We deploy servers such that memory addresses are from 0 to 299 and each server has 100 addresses. In other words:

- Server with index 0: addresses [0, 99] (Windows)
- Server with index 1: addresses [100, 199] (Linux)
- Server with index 2: addresses [200, 299] (Linux)

The addresses and ports used by servers are statically known and included in a `.env` file of environment variables. A user acting as an administrator manually puts up the Servers on their respective internet addresses and then the System is ready for Clients.

Component Design

System architecture is reflected in the component design details presented below. Also, the class diagram shown later depicts some of the interactions between these components (in the class diagram we treat them as the corresponding classes).

1. ClientLogic: A component that hides communication details and provides an interface that clients may use to perform operations on the Distributed System.
2. Client: A user interface that uses the interface provided by the `ClientLogic` component. A client may be one of our Python clients, Java clients or a Python ClientWrapper which allows users to use either Python or Java through a Python module.
3. Server (Node):
    a. MemoryItem: a single MemoryItem that represents data stored at a single memory address in our System.
    b. LockItem: a component that provides a locking abstraction to be used for managing memory accesses consistently.
    c. MemoryManager: a component that manages the operations performed on the main memory of a Node. It, also, implements the synchronization logic providing appropriate locking for its data items (MemoryItem components).

    d. Cache: a component that manages the functions of the local cache in each Node. It uses a very simple replacement algorithm: if the shared memory has size N memoryItems then memory address M is matched with position M mod N in the cache.
    e. Communication: the server handles communication with clients and serves them. It also handles situations where it needs to contact a different Server.
4. Network:
    a. Communication: message passing and data sharing between Nodes and Clients is done using TCP sockets. When a message M is to be sent. The sender first sends a message of fixed size (HEADER_LENGTH) which informs the receiver about the actual size of M, then M is sent. Messages are in JSON format. We already showcased the types of messages returned by Servers, so now we will show the messages sent from Clients to Servers: {"type": str, "args": list }. "type" informs the server of what operation to perform and "args" (if present) contains the arguments to be passed to that operation.
    b. Topology: A mesh topology. Every Server can communicate freely with every other Server.

Further details of the System Architecture

Our system is influenced by the 'Plus' system in the DSM lecture notes. When a client makes a write request to a memory address that is not local to its Server, then that server forwards this request to the host server of the memory address. The host server then initiates the process of updating copies (write – update) having added the server that made the request in the copyholder list. We use the idea of copyholder chain from the 'Plus' system where each copyholder – Server is responsible for communicating the next one, thus avoiding too much network traffic on the host server. One difference is that our Servers are rather 'stateless' and do not retain information about pending updates. Thus, when the write – update sequence is completed instead of receiving an acknowledgement from the last Server in the chain, each Server sends an acknowledgement to its predecessor until it reaches the host server. If a server in the chain is corrupted (not able to communicate or somehow down) then the original Server is informed and will remove ALL copyholders that could not be reached from the copyholder list. If there are none remaining, then the item becomes exclusively owned again.

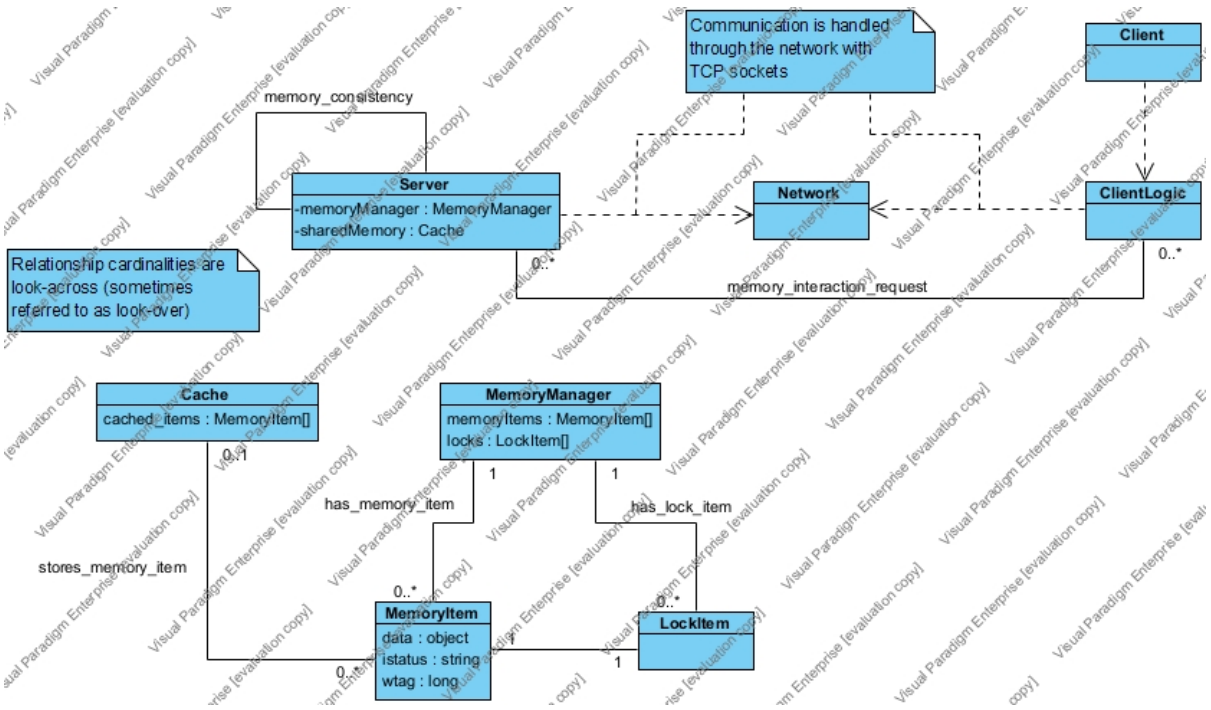
Reads to non-local memory addresses also update the local cache. Reads to non-local memory addresses that are in the cache also contact the host Server of this memory address in order to acquire the lock for it.

Locks are single – reader/single – writer for ease of implementation and this helps us enforce a consistency model on our data items.

State of Servers

The only information retained about state is in the Memory Items held by the Memory Manager and the Shared Memory. Which are the “istatus” of an item and the tags “wtag” and “ltag”. The Memory Manager also retains the copy holder chain. Apart from that, our Servers are stateless which makes implementation easier but has a greater communication overhead since our system makes a lot of connections and shares a lot of metadata each time.

Class Diagram



Relationships:

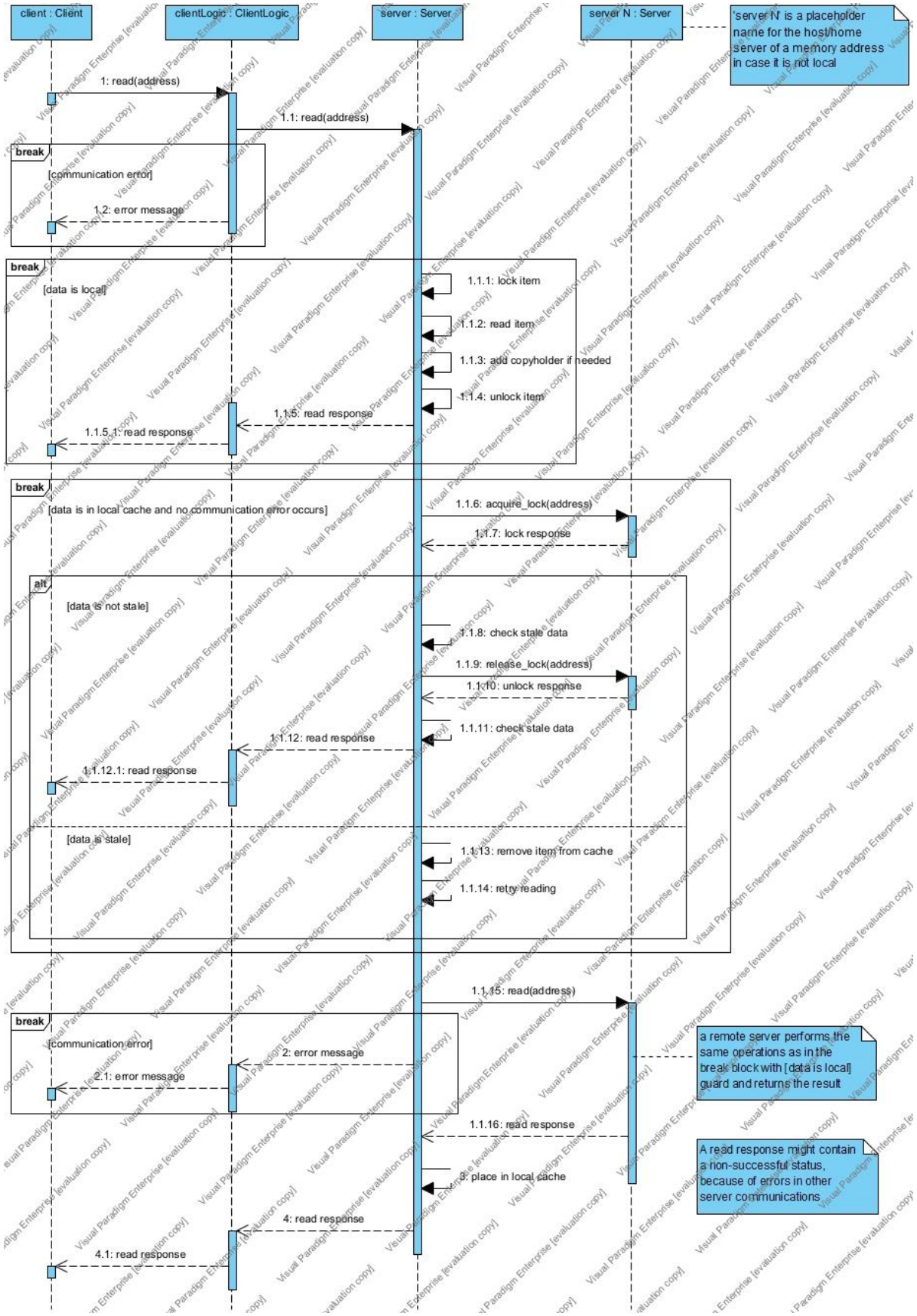
- memory_interaction_request: User requests some memory operation (read, write and might cause updates).
- memory_consistency: Nodes interact with other nodes to maintain memory consistency between them.
- has_memory_item: A memory manager controls the behaviour of a specific memory item.
- has_lock_item: A memory manager has a lock item, which controls locking on a memory item.
- stores_memory_item: Cache memory stores locally memory items obtained from remote Servers.

Protocols used have been mentioned in the 'Further details of the System Architecture' section.

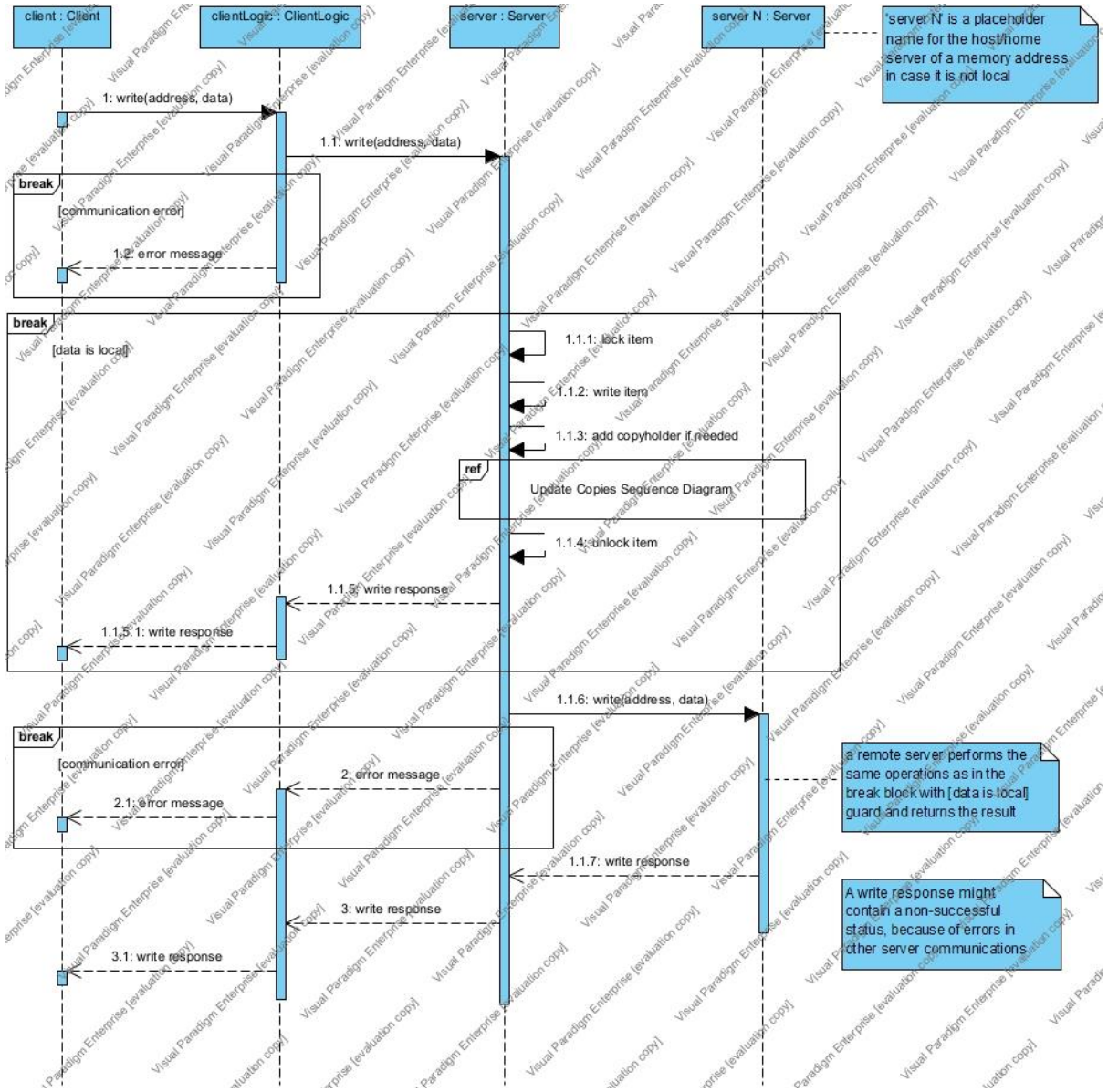
Sequence Diagrams

We show the sequence diagrams of read, write and update cache operations, since these are the main focus of the project.

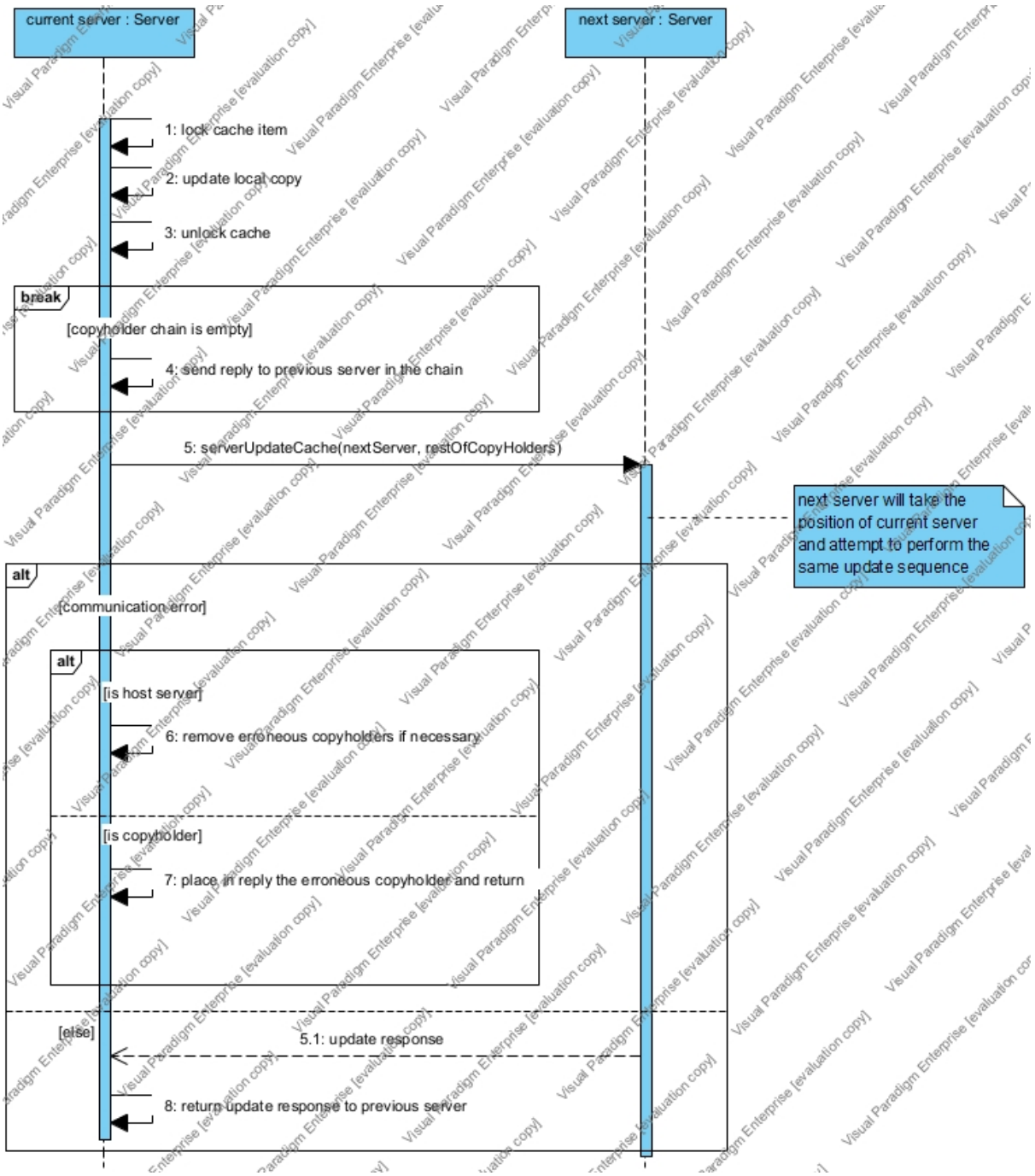
Read Sequence Diagram



Write Sequence Diagram



Update Copies Sequence Diagram



Of course, the host Server does not have a "previous server" and does not perform those actions/send those messages.

Comments

A few comments on the system.

First, let us comment on the read() function. Especially, the case when a remote memory item is already cached locally. We only acquire and release the lock from its host server instead of doing a whole read request to it. The idea is that if the user wants to have large data items then it might be too costly/slow to retransfer the whole item through the network. Thus only the locking information is transferred over the network to ensure consistency.

As explained read(), write(), lock() and unlock() return a "wtag" (last write tag) and an "ltag" (last lock tag). The "ltag" is incremented every time a lock is acquired and/or released and the "wtag" is incremented every time the memory address is written to. Thus the "wtag" values of a specific memory address enforce a sequence on reads and writes on a specific memory address (although it does not give us a happened-before relation between reads, since all reads between two writes have the same "wtag" value). Since, every time we read or write the lock is acquired and released, the "ltag" values of reads and writes enforce an order on all read and write operations on the same memory address. A client of this DSM may use this information to perform their own external synchronization.

One problem we have to combat is the possibility that a lock is acquired by a remote Server that becomes unavailable and thus the lock is never released. To fix this issue external locks are given with "leases" meaning that the lock will automatically be unlocked after a certain amount of time. Our Java code deploys a Timer object that schedules all these "automatic release" tasks, whereas our Python implementation uses a more software heavy approach deploying an individual "automatic release" thread after each lock is remotely acquired. The code is carefully constructed to deal with further issues that may arise, such as a remote Server attempting to re-release a lock after the timer has expired (the "ltag" is used ensure that a release request is valid) or a remote server attempting to return stale data from its cache to the Client. Using these tags allows our code to handle communication failures/server crashes without sacrificing Consistency of memory items stored in memory addresses.

We address the fact that the 'data' field was intentionally left untyped. This allows the user of this system to store any type of data and any size of data in the memory items of our memory addresses. Effectively, one can consider 'data size' to be the 'page size' of this Distributed Shared Memory System. Later we test the system for different sizes of data.

Implementation Issues

- Structure: data is wrapped in MemoryItem objects but the data field of these objects, which is the field where the actual data is stored, can be of any type
- Granularity: depends on the user's demands. The system does not impose any size restrictions
- Access Methods: Servers can be accessed by TCP connections and JSON messages

- Synchronization primitives: The system uses locks for each Memory Item. If a lock L corresponds to MemoryItem I, then the Server (Node) that manages I, also manages accesses to L. Locks are single-reader/single-writer and with the help of “wtag” and “ltag” fields provide consistency to memory accesses across the whole system
- Replication and coherence semantics: Coherence semantics already discussed. Replication occurs only in the context of cached data items, the behaviour of which is explained elsewhere in the report.
- Scalability: the project is designed in quite a scalable format, since the System’s size, Node count and other parameters depend on values in the `.env` file. However, the System is not dynamically scalable, meaning: once it has been created and the Nodes are running, new memory addresses cannot be introduced/properly managed by the System.
- Heterogeneity: System runs on Windows, Linux and Nodes/Clients are implemented in Python and Java. Communication is done through TCP sockets and JSON objects which is a universal and simple communication method.
- Fault tolerance: If a Node becomes unavailable, its memory addresses become inaccessible (even if previously cached) because we can’t be sure if it is still running and other Clients are connected to it (we would risk Inconsistency then) or if it has completely failed. The System appears quite fault tolerant to network errors and has a built-in mechanism for locking errors. Crashed Servers must be manually restarted and their memory is reset. The System’s memory is not persistent.
- Security: data security ensured by the locking protocol and the high-level memory abstraction used in the memory’s representation.

Differences from Original Concept

The original Use Cases have changed. System ‘start up’ and ‘exit’ are now performed manually by executing the corresponding programs that control Servers (Nodes). Memory allocation is no longer needed; the system has a fixed amount of memory known to all clients and all clients share the same memory addresses, thus freeing memory is not needed either. Health – check is not included, but system status is implicit now, meaning if the System becomes partitioned or some Node fails in some way then a user will not learn of this situation unless they try to access data items hosted or cached (only when doing writes) by this Node. In this case they simply receive an error message. Also, new Use Cases were included such as connect, disconnect and the auxiliary ones.

There is no explicit Synchronisation Manager, Fault Manager, Mapping Manager or System Monitor now. Let us see why:

- Synchronisation Manager: the operations of this Manager are now included in the Memory Manager and the associated locking features it provides.
- Mapping Manager: we assume the IP addresses, PORTs of Nodes and Memory size are known and static. Memory addresses are given to nodes implicitly, by the way that they are created from the Server Application scripts and thus a Mapping Manager is no longer needed.

- System Monitor: Our Servers use TCP sockets for communication and each time a new request needs to be served another socket is created. Health – checking is thus performed implicitly by serve requests when they attempt to communicate with other Servers and have to handle errors that result from failure of communication.
- Fault Manager: We have already discussed about why no System Monitor exists in our current system or why the Health – Check operation is not present explicitly. The Fault Manager need not exist as its functionality is included in the Server's error handling. We mention the CAP theorem now: when our DSM system becomes Partitioned, it preserves Consistency, but does not provide Availability meaning that Clients can perform operations on memory addresses that belong to Servers that are still accessible, but are denied such operations on unavailable Servers (even if that memory item is locally cached). In this way, the System remains partly available to the Client, but provides consistent memory accesses, while it remains partitioned and until all nodes become available again.

Test Plans

- `test.py`: This file contains some basic testing. It tests:
    - Connecting clients to server
    - Writing to local addresses
    - Reading from local addresses
    - Acquiring and releasing local locks
    - Writing to remote addresses
    - Dumping cache of Servers through clients
    - Reading from local addresses
    - Reading from remote addresses
    - Acquiring and releasing remote locks
    - Disconnecting servers

  Using the output logs of Servers (in the terminal output) and the output of the test we can check if any unexpected behaviour occurs.
    - It also tests if remote reads and remote writes update the cache appropriately
    - If stale cache entries are updated appropriately
    - If corrupted copyholder are handled correctly (removal from copyholder chain)
- `test_forgotten_locks.py`: This file tests if locks that were acquired by a Client but never released are actually usable after the release timeout expires.
- `test_concurrent.py`: Tests if all reads after a write return the same data. Test done both on reads that are local to the data item and ones that are remote.
- `test_times.py`: A file that provides us some performance metrics for our System.

The tests that were originally planned for this Distributed Shared Memory System have been adapted as shown above to better test its new functionality and testing on Components that no longer exist has been removed.

Performance

Metrics and results from time measurements:

```
PS > python .\test_times.py -reps 100
Testing basic functionality
Put all the server up and running and press enter to continue
--------------------------------------------------
Testing with small data
--------------------------------------------------
Testing serial reads to the same memory address
Time taken by test_serial_reads: 7.9837 seconds
Testing serial writes to the same memory address
Time taken by test_serial_writes: 9.9880 seconds
Testing concurrent reads to the same memory address
Time taken by test_concurrent_reads: 3.2878 seconds
Testing concurrent writes to the same memory address
Time taken by test_concurrent_writes: 5.1035 seconds
Testing serial reads to random memory addresses
Time taken by test_random_reads: 5.2412 seconds
Testing serial writes to random memory addresses
Time taken by test_random_writes: 7.6753 seconds
Testing random concurrent reads
Time taken by test_random_concurrent_reads: 0.2718 seconds
Testing random concurrent writes
Time taken by test_random_concurrent_writes: 0.2861 seconds
--------------------------------------------------
Testing with large data
Testing serial reads to the same memory address
--------------------------------------------------
Time taken by test_serial_reads: 7.9104 seconds
Testing serial writes to the same memory address
Time taken by test_serial_writes: 9.9521 seconds
Testing concurrent reads to the same memory address
Time taken by test_concurrent_reads: 3.4530 seconds
Testing concurrent writes to the same memory address
Time taken by test_concurrent_writes: 5.1232 seconds
Testing serial reads to random memory addresses
Time taken by test_random_reads: 5.0637 seconds
Testing serial writes to random memory addresses
Time taken by test_random_writes: 9.3963 seconds
Testing random concurrent reads
Time taken by test_random_concurrent_reads: 0.2665 seconds
Testing random concurrent writes
Time taken by test_random_concurrent_writes: 0.3481 seconds
> python .\test_times.py -reps 100
Testing basic functionality
Put all the server up and running and press enter to continue
--------------------------------------------------
Testing with small data
--------------------------------------------------
Testing serial reads to the same memory address
Time taken by test_serial_reads: 0.2684 seconds
Testing serial writes to the same memory address
Time taken by test_serial_writes: 5.0540 seconds
Testing concurrent reads to the same memory address
Time taken by test_concurrent_reads: 0.1463 seconds
Testing concurrent writes to the same memory address
Time taken by test_concurrent_writes: 5.0620 seconds
Testing serial reads to random memory addresses
```

```
Time taken by test_random_reads: 1.4646 seconds
Testing serial writes to random memory addresses
Time taken by test_random_writes: 2.4805 seconds
Testing random concurrent reads
Time taken by test_random_concurrent_reads: 0.1431 seconds
Testing random concurrent writes
Time taken by test_random_concurrent_writes: 0.1688 seconds
--------------------------------------------------
Testing with large data
Testing serial reads to the same memory address
--------------------------------------------------
Time taken by test_serial_reads: 0.2255 seconds
Testing serial writes to the same memory address
Time taken by test_serial_writes: 5.0564 seconds
Testing concurrent reads to the same memory address
Time taken by test_concurrent_reads: 0.1515 seconds
Testing concurrent writes to the same memory address
Time taken by test_concurrent_writes: 5.1000 seconds
Testing serial reads to random memory addresses
Time taken by test_random_reads: 1.4176 seconds
Testing serial writes to random memory addresses
Time taken by test_random_writes: 3.3028 seconds
Testing random concurrent reads
Time taken by test_random_concurrent_reads: 0.1487 seconds
Testing random concurrent writes
Time taken by test_random_concurrent_writes: 0.2216 seconds
$ python3 test_times.py -reps 100
Testing basic functionality
Put all the server up and running and press enter to continue
--------------------------------------------------
Testing with small data
--------------------------------------------------
Testing serial reads to the same memory address
Time taken by test_serial_reads: 0.3095 seconds
Testing serial writes to the same memory address
Time taken by test_serial_writes: 5.0777 seconds
Testing concurrent reads to the same memory address
Time taken by test_concurrent_reads: 0.1483 seconds
Testing concurrent writes to the same memory address
Time taken by test_concurrent_writes: 5.0430 seconds
Testing serial reads to random memory addresses
Time taken by test_random_reads: 1.6585 seconds
Testing serial writes to random memory addresses
Time taken by test_random_writes: 2.6847 seconds
Testing random concurrent reads
Time taken by test_random_concurrent_reads: 0.1746 seconds
Testing random concurrent writes
Time taken by test_random_concurrent_writes: 0.1911 seconds
--------------------------------------------------
Testing with large data
Testing serial reads to the same memory address
--------------------------------------------------
Time taken by test_serial_reads: 0.2172 seconds
Testing serial writes to the same memory address
Time taken by test_serial_writes: 5.0716 seconds
Testing concurrent reads to the same memory address
Time taken by test_concurrent_reads: 0.1365 seconds
Testing concurrent writes to the same memory address
Time taken by test_concurrent_writes: 5.0342 seconds
Testing serial reads to random memory addresses
Time taken by test_random_reads: 1.5995 seconds
```

```
Testing serial writes to random memory addresses
Time taken by test_random_writes: 3.5592 seconds
Testing random concurrent reads
Time taken by test_random_concurrent_reads: 0.1509 seconds
Testing random concurrent writes
Time taken by test_random_concurrent_writes: 0.2601 seconds
```

Comments on performance

- First run: (original configuration)
    - 1 Java Node in Windows
    - 1 Java Node in WSL
    - 1 Python Node in WSL
    - 100 Python Clients in Windows
- Second run:
    - 1 Java Node in Windows
    - 2 Python Nodes in WSL
    - 100 Python Clients in Windows
- Third run:
    - 2 Java Nodes in Windows
    - 1 Python Node in WSL
    - 100 Python Clients in WSL

We see that the communication between Java nodes and the Java nodes' performance when running in the WSL system is significantly slower and impacts runtime quite a lot. Apart from that, serial operations are, also, significantly slower than concurrent ones. Concurrent random operations showcase the biggest performance difference from any other, finishing very quickly. This behaviour can be attributed to our single-reader/single-writer locking mechanism, which greatly favours accesses in different memory addresses and hinders (concurrent) accesses to the same address.

We would also like to address the maximum connection limit which exists in Java Servers and leads to connection refusals when testing with a lot of Clients. Such a limit is not present in Python Servers.

We see that the performance is not greatly impacted by the data size ('small data' refers to single character data, while 'large data' refers to 1024 character data)

Notes

Project software structure and usage explained in the `README.md` file. We present the `README.md` below:

README

EDCS WUT Project

Authors

Dimitrios Georgousis (K-7729)

Gabriel Paic

Description

Topic: 13 write – update. Implement distributed memory with read and write access, using write-update protocol. We present a concept design for the above distributed system which reflects our current vision of implementation. The aim of the project is educational.

Installation

Clone this repo. And follow usage instructions.

Environment

Windows (11 Pro 64-bit):

```
PS > java -version
java version "1.8.0_411"
Java(TM) SE Runtime Environment (build 1.8.0_411-b09)
Java HotSpot(TM) Client VM (build 25.411-b09, mixed mode, sharing)
PS > python --version
Python 3.12.1
PS > pip show jpype1
Name: JPype1
Version: 1.5.0
Summary: A Python to Java bridge.
Home-page: https://github.com/jpype-project/jpype
Author: Steve Menard
Author-email: devilwolf@users.sourceforge.net
License: License :: OSI Approved :: Apache Software License
Location: <somewhere>
Requires: packaging
Required-by:
```

Linux:

```
$ lsb_release -a
```

```
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.4 LTS
Release:        22.04
Codename:       jammy
$ java -version
openjdk version "11.0.22" 2024-01-16
OpenJDK Runtime Environment (build 11.0.22+7-post-Ubuntu-
0ubuntu222.04.1)
OpenJDK 64-Bit Server VM (build 11.0.22+7-post-Ubuntu-0ubuntu222.04.1,
mixed mode, sharing)
$ python3 --version
Python 3.10.12
$ pip show jpype1
Name: JPype1
Version: 1.5.0
Summary: A Python to Java bridge.
Home-page: https://github.com/jpype-project/jpype
Author: Steve Menard
Author-email: devilwolf@users.sourceforge.net
License: License :: OSI Approved :: Apache Software License
Location: <somewhere>
Requires: packaging
Required-by:
```

Project Structure

```
$ tree
├── java_code
│   └── edcs
│       ├── edcs.iml
│       ├── out
│       │   ├── artifacts
│       │   │   ├── client_app_jar
│       │   │   │   └── client-app.jar
│       │   │   └── server_app_jar
│       │   │       └── server-app.jar
│       └── src
│           └── main
│               └── edcs
│                   ├── application
│                   │   ├── ClientApp.java
```

```
│                   │   └── ServerApp.java
│                   └── project
│                       ├── Cache.java
│                       ├── ClientLogic.java
│                       ├── CommUtils.java
│                       ├── GlobalVariables.java
│                       ├── LockItem.java
│                       ├── MemoryItem.java
│                       ├── MemoryManager.java
│                       ├── Server.java
│                       ├── TimeUtils.java
│                       ├── Tuple.java
│                       └── Tuple2.java
├── python_code
    ├── cache.py
    ├── client.py
    ├── client_logic.py
    ├── client_wrapper.py
    ├── comm_utils.py
    ├── global_variables.py
    ├── memory_manager.py
    ├── memory_primitives.py
    ├── server.py
    ├── test.py
    ├── test_concurrent.py
    ├── test_forgotten_locks.py
    ├── test_times.py
    └── time_utils.py
```

We would advise you to first look at the python code and then the Java implementation. Commenting in the Python version is much more verbose. Nonetheless, important details are commented in the Java version too.

Explanations:

- `global_variables`: loads environmental variables from .env file
- `time_utils`: provides an interface used for timestamping write and lock tags in our code.
- `comm_utils`: implements the communication protocol between our TCP sockets (a message is sent in two parts: The first part is of fixed length and contains information about the length of the actual message and then the actual mesasge is sent)
- `memory_primitives`: contains `memory items` and `lock items` which are used by `memory_manager` and `cache` for storing and synchronization.
- `memory_manager`: handles the main memory accesses to a Node's memory addresses.
- `server`: uses a memory manager and a cache object internally. `Server` is synonymous to `Node` in this project. It also handles communication with clients by accepting their

connections and serving their requests but may also make requests to other servers through the `_get_from_remote()` method.
- `client_logic`: wraps the requests that a client may send to a server in a more user friendly way
- `client_wrapper`: allows one to wrap a Python class around either a Python `client_logic` object or a Java `ClientLogic` object. This class is used in testing and allows testing both Python and Java clients.
- `client`: simple client that connects to a server and performs operations inputted by the user
- `test*`: these files can be used for testing various behaviours of our system
- `ClientApp`: same as the Python `client` but for the Java implementation
- `ServerApp`: implements the main method of the Python `server` module, but in Java.

The Java code was written as an IntelliJ IDEA Java Project. The Java jar files which are used for servers and clients (see usage) were produces as Artifacts through IntelliJ in case you wish to reproduce them.

Usage

.env files

```
HEADER_LENGTH=64
FORMAT=utf-8
CONNECTION_TIMEOUT=10
LEASE_TIMEOUT=8
SERVERS=192.168.160.1:6000,192.168.170.134:6001,192.168.170.134:6002
MEMORY_SIZE=300
CACHE_SIZE=50
MAXIMUM_CONNECTIONS=400
SUCCESS=0
ERROR=1
INVALID_ADDRESS=2
INVALID_OPERATION=3
JAVA_JAR_FILE=../java_code/edcs/out/artifacts/server_app_jar/server-app.jar
```

A `.env` file such as this must be present in the directory from which we run Servers or clients. The `JAVA_JAR_FILE` variable is used when performing tests using the Java classes instead of the Python ones.

As mentioned in the `concept` the Servers' addresses and memory space are static and are set by the above environment variables.

Python code:

```
/python_code$ python3 server.py -h
usage: server.py [-h] -server SERVER
```

```
Start a server process

options:
  -h, --help      show this help message and exit
  -server SERVER  The index of the server in the list of servers
/python_code$ python3 client.py -h
usage: client.py [-h] [-server SERVER]

Client to connect to a memory server

options:
  -h, --help      show this help message and exit
  -server SERVER  The index of the server in the list of servers, if
this
                  option is missing connect to a random server
```

Java code:

```
/EDCS/java_code/edcs/out/artifacts/server_app_jar$ java -jar server-
app.jar -h
Usage: java -jar server-app.jar -server <SERVER_INDEX>
Start a server process

Options:
  -server <SERVER_INDEX>   The index of the server in the list of
servers
/EDCS/java_code/edcs/out/artifacts/client_app_jar$ java -jar client-
app.jar -h
Usage: java -jar client-app.jar -server <SERVER_INDEX>
Start a client process which connect to a memory server

Options:
  -server <SERVER_INDEX>   The index of the server in the list of
servers, if this option is missing connect to a random server
```

In general, it is advised and expected that you have the Servers up and running before connecting clients to them. For example: the proposed deployment is one Windows Java Server, one Linux Java Server, one Linux Python Server and Python cliens running on the Windows machine. To properly deploy this system one would execute something like the following:

- `java -jar server-app.jar -server 0` on Windows
- `java -jar server-app.jar -server 1` on Linux
- `python server.py -server 2` on Linux
- and run clients on the Windows machine.