Warsaw University of Technology

Faculty of Electronics and Information Technology

Dimitrios Georgousis

Gabriel Paic

Distributed Computing and Systems

Project Concept

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 will be 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).

Use Cases

- Start up: An operation that starts the system up, so it is operational.
- Exit: An operation that shuts the system down.
- Memory allocation: An operation that, when called, allocates memory from the distributed system to the process calling it.
- Free memory: An operation that, when called, frees memory that was allocated to the process calling it.
- Read: Given a (virtual) memory address in the distributed memory, read from it.
- Write: Write to a (virtual) memory address in the distributed memory.
- Health – check: An operation that allows the user to check the status of the system/nodes.

The system will have one user type that will be able to perform all operations on it. The above use cases will be provided either as CLI commands or as a library package to some programming language. This will be decided further down the development cycle.

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. User Interface: A process that lets the user interact with the distributed system. As mentioned earlier it will be either a CLI or a program library that allows the user to do all necessary operations on the system.
2. Node:
    a. Storage Engine: each node maintains a local storage system for its segment of the distributed system's memory. A node, also, locally stores copies of non-local memory that it attempts to access. If a program tries to read from/write from a non local memory address, it has to request for that memory block.
    b. Communication Manager: handles networking communications for read, write, update requests.

    c. Synchronisation Manager: this component will work with the 'Communication Manager' and utilize some kind of locking system to ensure data consistency. The nature of this component will be explored further during development of the software.
    d. Fault Manager: Answers to health – check requests for a specific node and, also, sends heartbeats to the 'System Monitor' to confirm that the node is operational. Further functionality might be included later during development.
3. Mapping Manager: provides each 'Node' with some segment of the DS's memory to be managed by the 'Storage Engine' of that 'Node'.
4. System Monitor: a component that answers to the user's health – check requests and collects the node heartbeats to manage the systems health. Requests to start and stop the system go through this component.
5. Network:
    a. Communication: A protocol for message passing and data sharing between 'Nodes'/'Mapping Managers' must be used. It will be TCP (or UDP depending on implementation). Message types are 'READ_REQUEST', 'WRITE_REQUEST', 'UPDATE_NOTIFICATION', 'SYSTEM_HEALTHCHECK' and 'HEARTBEAT' and they include necessary metadata for the system to be operational.
    b. Topology: Most probably a mesh topology to allow for easy communication between 'Nodes' and other Components.

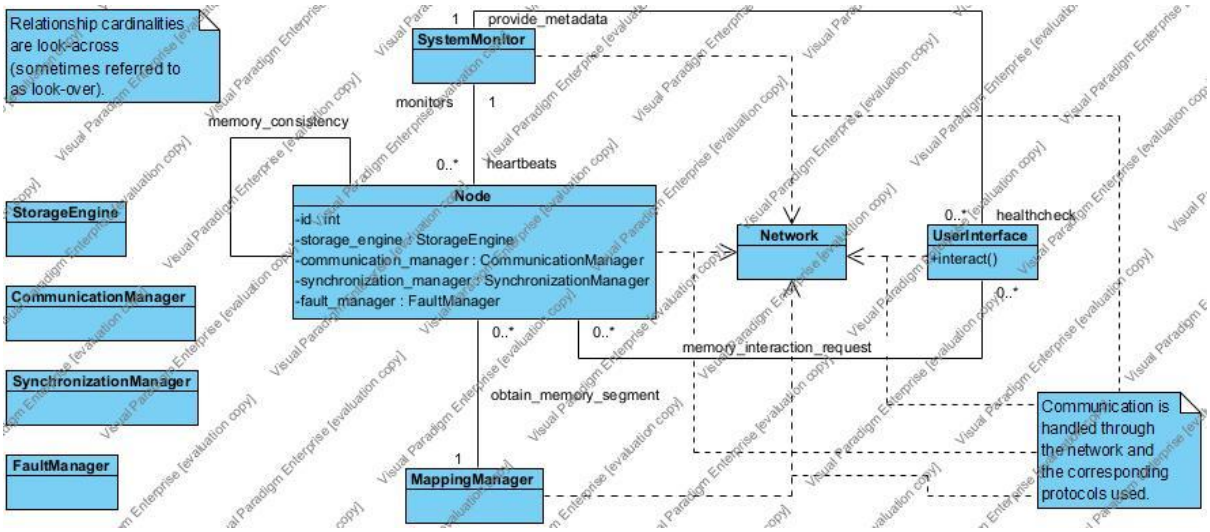
Further details of the System Architecture

1. Migration Algorithm: The algorithm for sharing non-local blocks. A very simple algorithm that simply sends the requested block over the 'Network' to the 'Node' that requested it.
2. Write – Update Protocol: This protocol is used by the component in 'Nodes' that handles write requests. When a write occurs to some memory address, the update is registered in the 'Node' where the address is local to and propagated to all other 'Nodes' that share this memory address.

As per the assignments request, this system should be developed to work on at least two different machines.

Some of the Components described above might be joined together to make the development easier, but it is too early to decide upon that yet.

Class Diagram



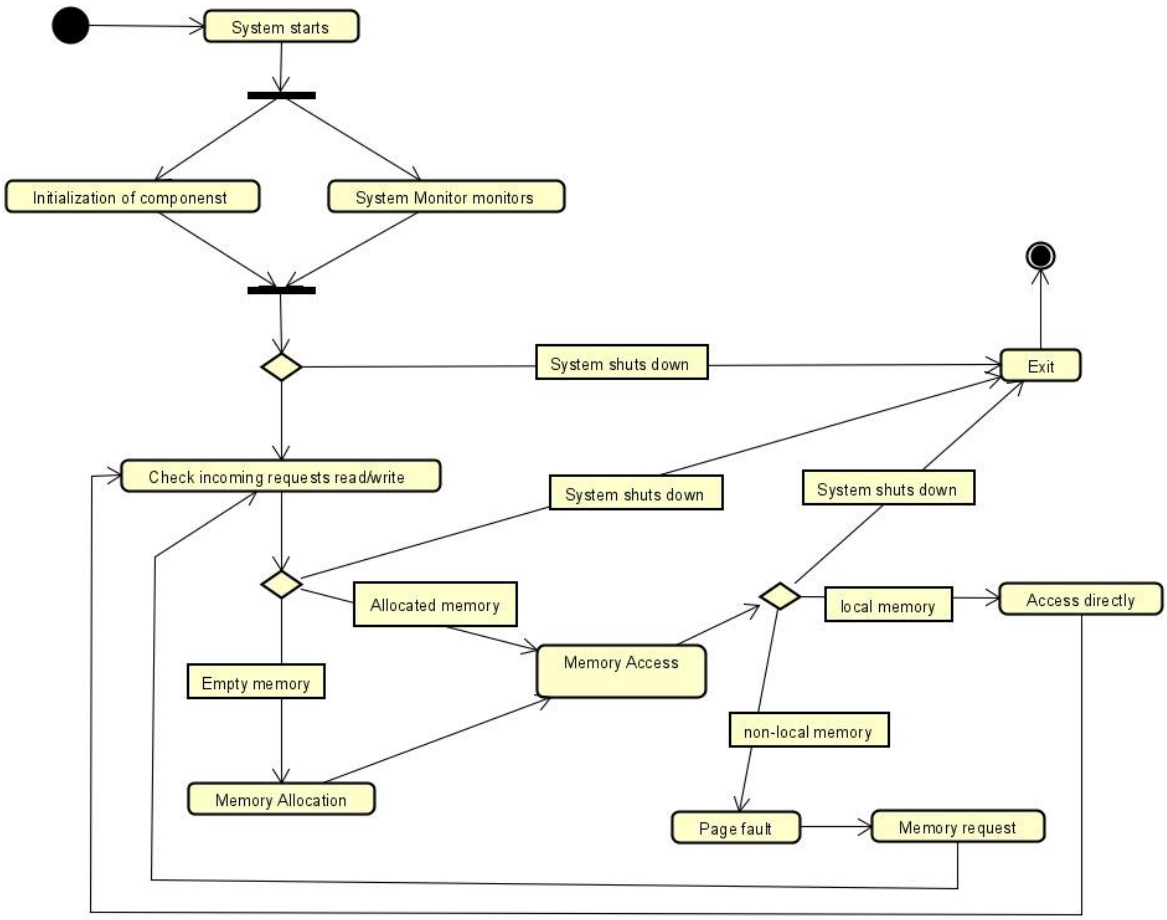
Relationships:

- monitors – heartbeats: 'System Monitor' monitors for 'heartbeats' received from 'Nodes'.
- provide_metadata – healthcheck: The user checks through the 'User Interface' the system's health.
- memory_interaction_request: User requests some memory operation (read, write and might cause updates).
- obtain_memory_segment: 'Node' obtains a memory segment of the distributed system's memory which it has to manage (to be more specific the 'Storage Engine' inside the 'Node' has to manage it).
- memory_consistency: Nodes interact with other nodes to maintain memory consistency between them.

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

Activity Diagram

We present an Activity Diagram that displays the general behaviour of our Distributed System.



Test Plans

Network test:

- Test 1: attempt to send through dummy 'Nodes' ('Nodes' with no actual functionality) all the needed messages/message types to check that the 'Network' protocol is working properly.

Node tests: these tests will focus on the functionality that a 'Node' should provide. Thus we will only use the interface that a node should provide to check that it works properly, while giving it a 'dummy' segment of the DS's memory.

- Test 2: perform a read()/write() request on an address of the segment given to the 'Node'. The request should occur normally.
- Test 3: perform a read()/write() request on an address of a different segment. The request should not affect local memory and should cause the 'Node' to send a request through the 'Network' for this memory address.

- Test 4: check that the 'Node' sends regular 'heartbeats' through the 'Network' when it is working properly.

Mapping Manager test:

- Test 5: Check that the manager segments the memory properly and gives each node an appropriate segment. The authentication that correct actions happened can be done by checking read()/write() actions on the system's 'Nodes'.

System Monitor tests:

- Test 6: Have a number, say N, of nodes active and check if the 'System Monitor' receives their heartbeats in a given time – window.
- Test 7: Have a number, say N, of nodes active and then cause one of them to fail (forcefully shut it down). The 'System Monitor' should detect a fault.
- Test 8: Attempt to shut down all nodes (the DS). The purpose of this test is to check if the exit()/stop() functionality works.
- Test 9: Attempt to start all nodes (the DS). The purpose of this test is to check if the start() functionality works.

User interface:

Through the 'User Interface' we can check the final system.

- Simply attempt to perform any of the 'Use Cases' through the user interface and check the results in two versions of the system:
    - When all Components are operational. The system should work properly.
    - When some Component fails. The system should maintain some of its functionality (maybe all of it in some cases) depending on what Component failed.
- Test for read()/write(): Attempt to read() or write() from a memory address of the DS that is on a segment in a different machine than the machine where 'User Interface' is deployed. Expected output: the system should handle this request without issues.

The above test plans cover testing over our system's intended functionality too.

Tests that may be added depending on development/implementation:

So far, our Mapping Manager and System Monitor do not take actions to balance loads or recover from failures. Recovery from failure of 'Nodes' is expected to happen manually. However, both load balancing and recovery are important so it is probable that they will be added to the final version of this software. Thus, appropriate testing for them will be included.