Programming project - Fault Tolerant CAD

In this programming project, you will translate a simple drawing tool to a distributed, fault-tolerant, collaborative environment that uses client/server architecture and supports Java clients.

It is recommended that you work in groups of 2-3 people on this assignment.

Overview

The centralized version of the tool is a simple program that allows you to draw circles, boxes and lines in a few different colors.

Interface

The buttons at the top of the tool window can be used to change the color and type of object that will be drawn. The current object template is displayed beneath the buttons. In the Standard Java version, you can draw objects by clicking and dragging the mouse in the drawing area (the black area below the button panel). To undo the last drawing, click the right mouse button.

Assignment goals/requirements

1. The overall goal of the assignment is to turn the tool into a collaborative, fault-tolerant environment across multiple platforms. Specifically, it should be possible to start several instances of the tool, some of which may be on different type machines (desktop), and all operations performed in one client should be seen in all other clients. In other words, the states of the drawing areas in all clients should be identical at all times. If a new client is started while other clients are running, it should immediately update its state to reflect that of the current collaborative drawing.
2. Moreover, the application should achieve *fault transparency* (a.k.a., fail operational) with respect to failures in the components that are the constituents of the service. The components should be fail-silent, that is, either they work or they do not respond. The solution should handle permanent omission failures in communication between components (e.g., lost messages) and in the components themselves (e.g., crashed replica managers or the front-end). You may make the single-failure assumption, that is, that a failure can be handled before the next failure (of the same type) occurs. N.B., this may require that redundancy in time must have a minimum interval between attempts to allow processes to crash, detect the crash and restart the process.
3. Overall architecture
    a. The service should be implemented following the client/server architecture style, where clients either have a RPC-based API encapsulating the message communication protocol with the service or an asynchronous message passing API. In the former case, you must handle exceptions and broadcasts as callbacks. The service should be implemented with a front-end and replica manager

processes. You may use either UDP or TCP as transport protocol between the clients and the front-end.

b. Let the front-end be a process that is state-less so that restarting it in case of a failure is as fast as possible. The front-end is identified by a port on a specific host. This data should be found in a configuration file for launching the application. A state-less process contains no information except the host and port where the primary replica manager is found.
c. The replica managers is a process each. The port numbers and hosts of the replica managers are assumed to read from a configuration file[1]. The replica managers must agree upon who is the primary where an identity is employed as the arbitration between processes in elections. This election must be held as soon as there are no primary; either in the beginning or at some point when it is detected that the primary is lost (you can use the same mechanism for detecting the loss of a primary and for startup).
d. To start the front-end or replica managers, you can use launcher processes that are running on the hosts. A draft launcher service is provided (SimpleProgramLauncher), note that this must be extended for you needs. Normally, the operating system is configured for this task, but this require rights that are inaccessible in many situations (e.g., in Linux, the init daemon process is configured to start processes, monitor them and, if required, restart the process. Do not overcomplicate the launching, the only thing the launcher process should do is to replicate the operating system functionality; read the configuration file, start the process and monitor (via the Process.waitFor()) if the process crashes. If the process crashes, then restart it as soon as possible. Note that the launcher can monitor several processes, you need one thread per process. Further, the launcher should essentially be as state-less as possible to avoid complexity.
e. Assume that the clocks are synchronized according to the NTP. Assume that the time granularity is 5 ms, that is, if two events occurs within 5 ms of each other, then they are considered to be concurrent.
f. The recommended election protocol is the Bully protocol, since it does not require the replica managers to be organized as a ring. You need to keep track of the state of each replica manager, for example: launched, integrated, primary, backup. Only when the replica manager is integrated (i.e., have recovered the remove objects from another replica manager that is either primary or backup) can it participate in an election.
g. Remote objects should be uniquely identified. For example, the UUID class can be employed to generate unique identifiers.
h. The front-end and the replica managers should communicate via jgroups protocol. Messages are multiplexed over the same channel, which means that threads listening to the channel must determine what to do. It is, for example, possible to get the UUID representing the message type directly out of a JSON object without converting it to a Java object and, thus, the front-end can redirect requests from clients to the appropriate replica manager (e.g., the primary) and vice versa

[1] The alternative is to specify a port that is used in an initializing multicast session during the booting of the service. We allow the much simpler protocol of using a configuration file here that is shared among the processes via a distributed file system.

without deserialization/serialization. Upon reception of a message, a replica manager decodes it and takes action upon the. One neat way of implementing actions is to encapsulate it in a method in the project-specific base class. This way, you do not need any switch statement in the receiving thread, it only calls action method on

4. Internal structure of processes
    a. To handle the protocol in the processes, it is recommended to employ a thread per socket in each process that is responsible for the protocol. You can, for example, use LinkedBlockingQueue<E>, to communicate between threads in the process according to some model of threading (1 thread per request, 1 thread per session, 1 thread per object). Then it is, for example, easy to handle timeouts with the Timer class where the TimerTask sends a message to the thread via the LinkedBlockingQueue<E>. Note that a LinkedBlockingQueue<E> can only have one receiver, but multiple senders. The use of LinkedBlockingQueue<E> enable safety as well as liveness properties (in contrast to wait and signal that only enable safety and guarantee safety in non-nested use).
    b. Organize your message types as classes with a superclass for all message types (e.g., named MessagePayload). That way, you can let the LinkedBlockingQueue<MessagePayload> contain objects of all message types. The only crux is when a message is read from the queue, it has to be cast to the proper class. One possibility, employed in the DistributedMessageSerialiation package, is to use the template method design pattern or the strategy design pattern, where the processing of the message is encapsulated into a class representing the message type. N.B., you typically have to pass a context to this processing method in order to enable a thread executing the method to reach necessary data structures. Further, it may be desirable to divide message types into request and reply types, since requests and replies share similar behavior in the processing.
5. Client API
    a. The client API should either be viewed as an RPC or RMI, where the message-passing is hidden or as an asynchronous message passing API. Asynchronous messages can be handled as exceptions as well as call-backs, that is, if an asynchronous message arrives that is not initiated via an RPC call, then a handler should be invoked to process the message. The client can register handlers in the API that take care of the asynchronous message. Typically, the handler should follow an interface specification.
6. Testing
    a. Employ the pict.exe tool to develop blackbox tests based on pairwise coverage. Inspiration for testing distributed systems are, for example:
        i. https://cacm.acm.org/magazines/2015/9/191167-testing-a-distributed-system/fulltext
        ii. https://cacm.acm.org/magazines/2016/2/197420-the-verification-of-a-distributed-system/fulltext
7. Jgroups configuration
    a. When you run jgroups, ensure that the flags –Dbind_addr=127.0.0.1 and -Djava.net.preferIPv4Stack=true when running the jgroups. The former switch is

there to enable communication on the localhost. The latter is required if both IPv4 and IPv6 are available. The jar files are available from jgroups.org.

Your design decisions should be well motivated in your report.

Getting started

First of all make sure that the provided source code compiles and run in your development environment.

- Java project and source code

Once you are familiar with the code, start thinking of the architecture and interfaces for the distributed application. *Remember to document in this step!* If you are unsure about anything in this step, feel free to discuss it with your teaching assistant or other students. When you have a clear idea of your architecture and have specified all interfaces, you can start implementing the application.

A few hints of where to start:

1. An overall design where different scenarios are described. N.B., make sure that exceptions are addressed as well as what the failure model
2. Consider the tutorial for jgroups.org.
3. Have a look at DistributedMessageSerialization and add message types (both request and reply)
4. Combine jgroups with DistributedMessageSerialization.
5. Study the SimpleProgramLauncher and how it works.

Examination

Each student team should submit a report, a presentation, well-commented source code (including any interface definitions), as well as working jar-files containing a runnable version of your desktop-client and server. Your will demonstrate your project in seminar form.

You are encouraged to discuss any problems you encounter with your fellow students outside your group too, but make sure to not co-operate between groups when writing the code or the report.

The deadline for the report, presentation and the code is available in SCIO. The seminar is in your schedule.

Deliverables

The project has the following deliverables:

- jar-files with your runnable solution
- source files for code review
- report
- presentation

Examination criteria

To pass the assignment, you must have working program that follows the specification under "Assignment goals" above, and it must be possible to run the application with the clients and/or any servers located on different machines. The report must be written properly - sloppy language and formatting is not acceptable. Badly written reports will be returned immediately without comments on report or code.

You must also present and demonstration your solution in the seminar.

Testing and verification are important aspects of building fault-tolerant applications. Since faults and extreme circumstances tend to be rare occurrences in a LAN environment, it is hard to test a system under such conditions unless they are explicitly addressed. In your report, you should have a thorough discussion about how you have verified and tested your program for fault tolerance and correctness.

Detailed criteria:

Goal/requirements	Pass	Fail
1	Fulfilled the requirement and achieved transparency. This should somehow be demonstrated via, for example, testing as well as analysis.	For example, frequent delays in updates, failures due to lost messages or crashed processes.
2	If replica managers or the front-end crashes, then the system will mask this and restart processes.	If either crashes of replica managers or the front-end is not masked away and there are, for example, delays that can be noticed by the user.
3.a	Fulfills the requirement.	For example, any of the following will lead to a fail: • The UDP or TCP/IP operations are directly visible to the application programmer. • Unnecessary visibility of failure handling protocols; for example, that the application program should handle lost

		messages or detection of lost processes. • The internal interfaces/classes are visible to the application programmer.
3.b	Fulfills the requirement.	For example, any of the following leads to a fail: • The front-end contains unnecessary state information that causes unnecessary delays in the startup/restarting of the process. • Exceptions that should be forwarded to the client are handled in the front-end. • The port/host is not read from a file in the launcher process.
3.c	Fulfills the requirement.	For example, any of the following leads to a fail: • The replica managers do not employ an election protocol to determine the primary. • The replica managers do not detect omissions of replica managers.
3.d	Fulfills the requirement.	For example, any of the following leads to a fail: • The launcher process does something more complex than reading the configuration file and start the appropriate programs given that the launcher runs on the host in the configuration file.
3.e	The system is based on this assumption.	There are evidence that part of the solution is not based on this assumption, for example, events are incorrectly ordered solely on the timestamp although they ought to be

		considered concurrent. If events are concurrent, then it is acceptable to arbitrate them via some unique identifier representing the source of the event.
3.f	The system fulfills the requirement.	Fails due to, for example: • Timeouts are not set properly with respect to the clock granularity or the round-trip time. • Replica managers that are not integrated can participate in elections.
3.g	Fulfills the requirement.	For example, any of the following implies a fail: • Remote objects are not uniquely identified. • Deleted remote objects can reappear. • A centralized solution is employed to generate unique identifiers.
4.a	Fulfills the requirement.	Fails due to, for example: • Some other mechanism that does not enable and support safety and liveness is employed rather than LinkedBlockingQueue. • Timeouts are handled by delays in the same thread that is responsible to the protocol. • A single-threaded solution is employed rather than a multi-threaded solution.
4.b	Fulfills requirement.	Fails due to, for example: • Disorganized message types. • Unnecessary non-uniform treatment of messages.

5	Fulfills the requirement.	Fails due to, for example: • The client API is not based on RPC. • Messages are visible. • Asynchronous messages are not handled by registered handlers.

Report requirements

When writing your report, make sure that you have covered at least the following things:

- A short description of the problem (maximum ½ page)
- An overview of the solution with the main design decisions covered and how the transparency properties are achieved.
- Detailed description of the solution emphasizing:
    - The protocol and message types involved between
        - Client and front-end
        - Front-end and replica managers
        - Replica managers with other replica managers
    - Design decisions should be well motivated and discarded alternatives should be addressed.
- A post-project reflection on the advantages and disadvantages of your solution.
- A discussion of possible future work and improvements. Would you have approached the problem differently if you could redo it from the beginning?
- A reflection on what you have learned during the project.
- Brief feedback on the assignment itself.

Presentation hints

Here are some requirements and hints regarding the seminar and demo session.

- The "demo devil" has a tendency to show up. Prepare some backup slides with screen shots to use if things fail on demo day.
- The presentation and demo should not be longer than 5-10 minutes. This means that you only have time for a few remarks in addition to demonstrating your program. No more than two (2) slides with information!
- On the actual seminar we will have two computers available so that you can prepare your demo at the same time as a presentation is underway. We will provide a presentation list.

You may use either Swedish or English for your report but the presentation should be in English. Regardless, use grammar- and spell checking tools!