2242-CSE-5306-004-DISTRIBUTED SYSTEMS
Project-2 Report

Jyothsna Pasupuleti -1002172750

Saikiranreddy Peddavootla -1002175284

I have neither given nor received unauthorized assistance on this work.

Project Overview

The distributed system being implemented in this project has a Transaction Coordinator (TC) who arranges transactions between various Participants. The TC and the Participants communicate with each other via the system using XML-RPC. During the preparation phase, the Participants vote on transactions, and the TC uses the votes to determine whether to move forward with the commit phase. The transaction process's failures and recoveries can be simulated by the participants.

Components

1. Coordinatator of Transactions (transaction_coordinator):
The fundamental element in charge of coordinating transactions in the distributed system is the Transaction Coordinator. It makes an XML-RPC server available for Participants to communicate with. The coordinator keeps track of the votes cast by Participants and oversees the preparation and commit phases of the transaction.

- **Features:**
  - begin_transaction(fail_case): Starts a new transaction with failing case (fail_case) specified.
  - prepare_participants(self, transaction_id, fail_scenario): Completes the transaction's prepare phase and collects participant votes.
  - commit_transaction(self, transaction_id, fail_scenario, proceed_to_commit): Based on the votes cast by Participants, the commit phase of the transaction is carried out.
  - emulate_failure(): Sleeps for a predetermined amount of time to simulate failure.
  - Logging: Captures messages and events pertaining to transactions.

2. Participant (transaction_participant):
In the distributed system, the Participant component represents a single participant. Participants participate in the transaction process and communicate with the Transaction Coordinator via XML-RPC.

- **Features:**
  - prepare_transaction(self, transaction_id): Provides a participant's vote while simulating the prepare stage of a transaction.
  - commit_transaction(self, transaction_id, proceed_to_commit): Based on the TC's choice, this function replicates the commit stage of a transaction.
  - emulate_failure(self, fail_scenario): Logs recovery if necessary and simulates failure by resting for a predetermined amount of time.
  - failure_check(self, fail_scenario): Returns the current timestamp and modifies the Participant's vote based on the supplied fail_case.

■ Logging: Documents communications and occurrences pertaining to participants.

3. ThreadedXMLRPCServer (transaction_coordinator and transaction_participant): The coordinator and participant modules both utilize this class to process XML-RPC queries in several threads, enabling simultaneous participant communication.

Initialization and Usage:

1.Participant Initialization (participant_node.py): In the distributed system, a single Participant is initialized via the tp_node.py script. Using the start function from the transaction_participant module, it launches the Participant after receiving the port number as a command-line input.

2.Initialization of the Transaction Coordinator (coordinator_node.py): The distributed system's Transaction Coordinator is initialized via the coordinator_node.py script. In addition to providing a list of Participant URLs to communicate with, it launches the coordinator on a designated port.

3.Transaction Initiation (start_transaction.py): This script calls the Transaction Coordinator's transact method, passing it a specific failure case as a parameter, to start a transaction in the distributed system.

coordinator/transaction_coordinator.py

1. **Coordinator Class:**

The distributed system's Transaction Coordinator is represented by the coordinator class. The prepare phase and the commit phase are the two primary stages it uses to manage transactions.

- __init__(self, participants, timeout=1):

  ■Provides the coordinator with an optional list of attendees at the beginning. Transaction_id: The transaction's ID.

  ■ The failure emulation timeout value.

- begin_transaction(self, fail_scenario):

  ■ Completes the prepare and commit steps in order to start a new transaction.

■ fail_scenario: An integer that represents the kind of failure scenario that should be simulated.

○ prepare_participants(self, transaction_id, fail_scenario):

■carries out the transaction's prepare phase.

■ transaction_id: The transaction ID.

■ Returns: False otherwise, true if every participant votes to commit.

○commit_transaction(self, transaction_id, fail_scenario, proceed_to_commit):

■ Completes the transaction's commit phase.

■ proceed_to_commit: A boolean representing whether or not the commit process should continue.

■ Returns: False otherwise, true if every participant commits successfully.

○ emulate_failure(self):

■ Sleeps for the duration of the timeout to simulate failure.

■ After the timeout, the current timestamp is returned.

2. ThreadedXMLRPCServer Class: This class allows the XML-RPC server to process RPC requests in multiple threads by extending the ThreadingMixIn and SimpleXMLRPCServer classes.

3. Initialize_coordinator Function: To handle RPC requests from participants, the start function launches the XML-RPC server and instantiates the Transaction Coordinator.

○ initialize_coordinator(port, t_participants =None):

■ Uses the provided participants list to instantiate the Transaction Coordinator.

■ port: The port to listen on.

■ t_participants: An inventory of the objects involved.

Logging

The snippet of code sets up logging to capture significant messages and events. The following are the logging settings:

- Logging level: INFO
- Log format: %(asctime)s [%(levelname)s] %(message)s
- Handlers: Both the console and a log file called tc.log receive log messages.

participant/transaction_participant.py

1.Participant Class:

A participant in the distributed system is represented by the Participant class.

- __init__(self, url, timeout: 1.):
  - Provides a participant's URL and an optional timeout value for failure emulation when they first join.
  - URL: The participant's URL.
  - timeout: The failure emulation timeout value (by default, 1.0).
- prepare_transaction(self, transaction_id):
  - Represents a transaction's prepare phase and yields the vote of the participant.
  - transaction_id: The transaction ID.
  - Returns: A string representing the vote ("yes" or "no").
- commit_transaction(self, transaction_id, fail_scenario, proceed_to_commit):
  - Recreates a transaction's commit phase by using the coordinator's choice.
  - transaction_id: The transaction ID.

■ proceed: A boolean representing whether the commit process should continue.

■ Returns: A string representing the commit message (":commit" or ":abort").

- emulate_failure(self, fail_scenario) :

■ Sleeps for the duration of the delay to simulate failure, logging recovery if fail_case is 4.

■ fail_scenario: A numerical value designating the kind of failure scenario to replicate.

■ Returns: Following the timeout, the current timestamp.

- failure_check(self, fail_scenario):

■ Returns the current timestamp and modifies the participant's vote based on the supplied fail_scenario.

■ fail_scenario: An integer that represents the kind of failure scenario that should be simulated.

■Returns: The most recent timestamp.

2.ThreadedXMLRPCServer Class:
To handle RPC requests in several threads, the ThreadedXMLRPCServer class extends the ThreadingMixIn and SimpleXMLRPCServer classes.

3.Initialize_participant Function:

The XML-RPC server is started via the start function, which also instantiates a participant to process RPC requests from the Transaction Coordinator.

- initialize_participant(port=8000):

■ Creates a participant using the host and port that have been supplied.

■ port: The port (8000 by default) to listen on.

setup_logger.py

This class snippet configures logging to record important events and messages. The logging settings include:

- Logging level: INFO
- Log format: %(asctime)s [%(levelname)s] %(message)s
- Handlers: Log messages are written to both a log file named participants.log and the console.

coordinator_node.py

1.Transaction Coordinator and Participants:

XML-RPC client proxies (xmlrpc.client.ServerProxy) are used to build the Transaction Coordinator and Participant objects. The coordinator and participants can communicate with one another at a distance thanks to the proxies.

2. Starting the Distributed System:

The coordinator's start function. To instantiate the coordinator and launch the XML-RPC server, the coordinator module is called. The function accepts two arguments: a list of participant proxies (Ps) and the port number (8000). The two participant proxies (P1 and P2) are included in the list.

3. Coordinator and Participant Modules:

It appears that the code is presuming that the coordinator and participant classes have their own modules. The code starts with the import of the modules:

- coordinator.transaction_coordinator: Probably houses the coordinator's other functions in addition to the start function.
- participant.transaction_participant: Probably holds the definition of the Participant class.

Usage Example

1. Using XML-RPC client proxies, the code initializes two participant objects (P1 and P2), each of which points to a distinct URL (http://localhost:8001 and http://localhost: 8002).

2. The two participant objects (P1 and P2) are added to the list Ps.

3. The coordinator's start function. To build the Transaction Coordinator, the coordinator module is called. The list of participant proxies (Ps) is supplied as an input to the coordinator when it is started on port 8000.

4. Participants can send and receive RPC calls to take part in distributed transactions, and the coordinator can connect with them via the XML-RPC server that is supplied.

participant_node.py

1.Participant Initialization:

The start method is imported by the code from the participant/transaction_participant module, which is assumed to include the functionality needed to create and launch a participant.

2.Command-LineArgument:
The command-line arguments are retrieved by the script via the sys.argv list. When executing the script, the port number must be supplied as the first argument.

3.Starting the Participant:
The participant is initialized by calling the start function. The port number (portno) is an argument passed to the function.

Usage Example

Run the script from the command line with the required port number passed in as an input to start a participant:

participant_init_script.py 8001 in Python 3

The script will use the supplied port number (e.g., 8001) to create and start a participant by calling the start function from the transaction_participant module.

Please be aware that the code snippet that is provided is predicated on the assumption that there is a transaction_participant module that has the features required to initiate a participant. This sample does not display the participant module's precise implementation details.

begin_transaction.py

1.Transaction Initiation: To manage XML-RPC communication, the code imports the xmlrpc.client module. It sets up a client proxy with the URL "http://localhost:8000" to establish a connection with the Transaction Coordinator.

2.Command-Line Argument: The command-line arguments are retrieved by the script via the sys.argv list. When the script is run, an integer argument must be supplied. The transaction is started using this integer value as an input.

3.Transaction Start:
Using the TC client proxy, the script invokes the Transaction Coordinator's transact method, passing the integer argument as the fail_scenario parameter.

Usage Example

The script, Python3 transaction_init_script.py, must be run from the command line with the desired integer value passed as an input to start a transaction. 2.

After establishing an XML-RPC client proxy for the Transaction Coordinator, the script will invoke the coordinator's transact method and supply the value 2 as the fail_scenario argument to initiate the transaction.

Notice that the code snippet provided presupposes that the Transaction Coordinator has a function named transact that can handle the fail_scenario parameter to start a transaction in the distributed system, and that it is running and listening at "http://localhost:8000".

Conclusion

The distributed system that has been put into place, featuring a Transaction Coordinator, demonstrates how several Participants can coordinate transactions. Participants can communicate with the Transaction Coordinator using XML-RPC communication, and logging offers insightful information about system events and the transaction process. Initializing Participants and the Transaction Coordinator is made simple by the scripts that are given.

Additional implementation details for the Participant and Transaction Coordinator modules, along with connection with other distributed system components, are required to expand the project. To guarantee the system's resilience and dependability in practical situations, appropriate testing and error management must be taken into account. Incorporating security features like encryption and authentication would also be necessary to guarantee secure communication inside the distributed system.

LEARNT FROM THE PROJECT

1. Two phase distributed commit protocol.

2. Recovery Strategies.

3. Fault Tolerance.

ISSUES ENCOUNTERED:

1. Handling node crashes

2. Timeout tuning.