UNIVERSITY OF TRENTO

Department of Information Engineering and Computer Science

Distributed Systems 1

2021/09/03

Project Report

Carlo Corradini

223811

Nicoló Vinci

220229

Academic year 2020/2021

Contents

1 Introduction

Our project is named `banky` to reflect a simulated bank system where multiple customers (clients) make multiple operations (read/write) simultaneously with the restriction that all client's operations are made in a transaction keeping the system always consistent.
All the available `Java` code is documented via *Javadoc* to have a more comprehensive view of the overall logic and context.

1.1 The project

The project consists on developing a protocol to manage distributed transactions. The protocol must ensure strict serializability for successfully committed transactions and abort any inconsistent transactions. The optimistic approach has been considered to check if a transaction is consistent or not. The decision is taken at the end of the transaction checking the item version that a transaction wants to modify. A transaction can be handled by one coordinator that validates it applying a two phase commit protocol to ensure consistency between the various data stores.

1.2 GitHub Repository

The source code and the documentation of *banky* are available on `GitHub` at the following URL: `https://github.com/carlocorradini/ds1`

1.3 Scenario

In the project scenario there are three main components:

- `Clients`
  A client wants to perform some operations to read or write item in data stores. A transaction is composed by different operations. The client can not contact directly data stores to execute a transaction, but they interface to them via one of the possible coordinators.

- `Coordinators`
  The coordinator obtains the various read or write requests of a transaction made by a client and it forwards them to the corresponding data stores. The coordinator keeps track of transactions assigning to them a unique identifier. At the end of a transaction, the coordinator manages the validation phase exploiting the two phase commit protocol to ensure consistency among data stores.

- `Data Stores`
  A data store keeps a mapping between key and item. An item is composed by the value and the version. When a transaction will be successfully committed, the version of involved items will be incremented by one.

1.4 Assumptions

Some assumptions have been considered during the implementation of the system:

- Channels are reliable and FIFO.

- The strict serializability requirement is only enforced for committed transactions.

- One client can perform one transaction at a time. However, multiple clients can generate concurrent transactions.

- Coordinator and data servers can crash. Then, they will recover after a certain delay.
- Clients don not crash.
- Crashes occur only during validation or commit phase.

1.5 Logger

In the project all messages and errors are logged using *Log4j* 2: an industry standard.
Apache Log4j is a Java-based logging utility. It was originally written by Ceki Gülcü and is part of the Apache Logging Services project of the Apache Software Foundation. Log4j is one of several Java logging frameworks. [...] The Apache Log4j team has created a successor to Log4j 1 with version number 2[1].
Apache Log4j 2 is an upgrade to Log4j that provides significant improvements over its predecessor, Log4j 1.x, and provides many of the improvements available in Logback while fixing some inherent problems in Logback's architecture[3].

1.5.1 Log levels

Log levels are a simple means of classifying log messages based on the information contained. During the configuration phase (see next section), a level is chosen from those in the list below (ordered by importance) in which it is expected that the lower levels are ignored compared to the higher ones.
In the project, the minimum *log level* defined in the `Root` tag is set to **INFO**.

FATAL
: The `FATAL` level designates very severe error events that will presumably lead the application to abort.

ERROR
: The `ERROR` level designates error events that might still allow the application to continue running.

WARN
: The `WARN` level designates potentially harmful situations.

INFO
: The `INFO` level designates informational messages that highlight the progress of the application at coarse-grained level.

DEBUG
: The `DEBUG` level designates fine-grained informational events that are most useful to debug an application.

TRACE
: The `TRACE` level designates fine-grained informational events than the `DEBUG`.

ALL
: The `ALL` has the lowest possible rank and is intended to turn on all logging.

OFF
: The `OFF` has the highest possible rank and is intended to turn off logging.

Log level \ Output	FATAL	ERROR	WARN	INFO	DEBUG	TRACE
FATAL						
ERROR						
WARN						
INFO						
DEBUG						
TRACE						
ALL						
OFF						

Table 1.1: Correlation between log levels and output

1.5.2 Configuration

In the project we have configured *Log4j* 2 via **Automatic Configuration** where a configuration file, named `log4j2.xml`, is used during the initialization phase.
The configuration file is available under `src/resources/log4j2.xml`.
The configuration file defines the available `Appenders`. *Appenders are responsible for delivering LogEvents to their destination.*
Two *Appenders* are available:

1. **`ConsoleAppender`**
   ConsoleAppender writes its output to either System.out or System.err with System.out being the default target. A Layout must be provided to format the LogEvent.

2. **`FileAppender`**
   FileAppender is an OutputStreamAppender that writes to the File named in the fileName parameter. The FileAppender uses a FileManager (which extends OutputStreamManager) to actually perform the file I/O.
   The generated log file name is named *banky.log*.
   Note that on every new execution the old file is overwritten with the new one. Therefore, if you want to keep the old log file remember to save it to another location.

Both Appenders are used in the default `Root` logger. Thus, whenever we call the logger, the message is *redirected* either on both of the Appenders.
As written in the previous section, the default level is set to `INFO`. This can be easily changed by modifying the `Root`'s *level* attribute.
For more information about configuring *Log4j* 2, refer to `https://logging.apache.org/log4j/2.x/manual/configuration.html`.

2 Configuration

The project has a configuration file named `Config` used to customize the application's behavior.
The file is under: `src/main/java/it.unitn.disi.ds1.Config`.
The following is a list describing all configuration variables (omitting those that are calculable):

- `MODE`
  `type`: *Mode*
  Mode to use when the application starts.
  Currently two modes are available:

  1. `INTERACTIVE`
     After a *run* is completed, the user is always asked if it wants to verify the storage to check the consistency and if it wants to continue with another *run*.
     Note that this mode is intrinsically infinite since there is not an upper limit on the number of total *run*(s) the user can achieve.
  2. `AUTOMATIC`
     There is no user interaction until all *run*(s) have been completed. The total number of *run*(s) is configurable (see next config variable). After all *run*(s) have been completed, a message to the user is prompted asking if it wants to verify the storage to check consistency; after that it terminates.
     This mode is very useful for debugging and testing purposes to verify that after all *run*(s) the system (Data Store(s)) it's still consistent.

- `N_RUNS`
  `type`: *int*
  The number of *run*(s) to process.
  Note that this variable is used only when *MODE* (see above) is set to `AUTOMATIC`, otherwise it will be ignored.

- `N_DATA_STORES`
  `type`: *int*
  The number of `Data Store`(s) in the system.
  Note that this reflects the total number of available `Item`(s) and the maximum Item key (`MAX_ITEM_KEY`, see below).

- `N_COORDINATORS`
  `type`: *int*
  The number of `Coordinator`(s) in the system.

- `N_CLIENTS`
  `type`: *int*
  The number of `Client`(s) in the system.
  Note that this reflects the total number of concurrent transactions in a *run*.

- `MIN_SLEEP_TIMEOUT_MS`
  `type`: *int*
  Minimum sleep timeout in millisecond(s). This is used in combination with `MAX_SLEEP_TIMEOUT_MS` to simulate a random delay due to network communication.

- `MAX_SLEEP_TIMEOUT_MS`
  `type`: *int*

Maximum sleep timeout in millisecond(s). This is used in combination with `MIN_SLEEP_TIMEOUT_MS` to simulate a random delay due to network communication.

- `TWOPC_COORDINATOR_TIMEOUT_MS`
  type: *int*
  2PC timeout in millisecond(s) when a `Coordinator` sends the vote request message (`TwoPcVoteMessage`) to the `Data Store`.

- `TWOPC_DATA_STORE_TIMEOUT_MS`
  type: *int*
  This timeout is used for two different purposes:

  1. 2PC timeout in millisecond(s) when a `Data Store` sends the vote response message (`TwoPcVoteResultMessage`) to the `Coordinator`.
  2. 2PC timeout in millisecond(s) when the `Data Store` is waiting the decision message (`TwoPcDecisionMessage`) from the `Coordinator`.

- `TWOPC_RECOVERY_TIMEOUT_MS`
  type: *int*
  2PC timeout in millisecond(s) to wait before recovering from a crash.

- `CRASH_COORDINATOR_ON_RECOVERY`
  type: *boolean*
  Crash the `Coordinator` during the recovery phase.
  Use this crash with caution since it produces cyclic crashes in the `Coordinator` causing a general blocking of the system.

- `CRASH_COORDINATOR_VOTE_FIRST`
  type: *boolean*
  Crash the `Coordinator` after sending the first 2PC vote request message (`TwoPcVoteMessage`) to the `Data Store`(s) involved in the transaction.

- `CRASH_COORDINATOR_VOTE_ALL`
  type: *boolean*
  Crash the `Coordinator` after sending all 2PC vote request message (`TwoPcVoteMessage`) to all the `Data Store`(s) involved in the transaction.

- `CRASH_COORDINATOR_DECISION_FIRST`
  type: *boolean*
  Crash the `Coordinator` after sending the first 2PC decision message (`TwoPcDecisionMessage`) to the `Data Store`(s) involved in the transaction.

- `CRASH_COORDINATOR_DECISION_ALL`
  type: *boolean*
  Crash the `Coordinator` after sending all 2PC decision message (`TwoPcDecisionMessage`) to all the `Data Store`(s) involved in the transaction.

- `CRASH_DATA_STORE_VOTE`
  type: *boolean*
  Crash the `Data Store` before sending the 2PC vote response message (`TwoPcVoteResultMessage`) to the `Coordinator` involved in the transaction.

- `CRASH_DATA_STORE_DECISION`
  type: *boolean*
  Crash the `Data Store` before receiving the 2PC decision message (`TwoPcDecisionMessage`) sent by the `Coordinator` involved in the transaction.
  Note that the crash is triggered right after the `Data Store` has sent the 2PC vote response message (`TwoPcVoteResultMessage`) to the `Coordinator` involved in the transaction.

3 Project structure

The overall project is in package `it.unitn.disi.ds1`
Inside are present multiple packages and two Java classes:

- **`actor`**
  Actors have been defined in this package. There is an abstract class called *Actor.Java* which includes the basic functionalities such as the actor id, the mapping between transaction and final decision, the timeout transactions and the crash state. Then, *Client.Java*, *Coordinator.Java* and *DataStore.Java* extend it.

- **`etc`**
  Two custom classes and one enumeration have been defined in this package. Each of them overrides the method *toString* returning a Java *Gson* instance. The *ActorMetadata.Java* class emulates a Java pair. Indeed, it has only two attributes: *id* and *ref*. Hence, this class associates an actor id to its reference. The *Decision.Java* enumeration defines two possible decisions: ABORT or COMMIT. So, it can be used during the validation phase by a *Coordinator* to take the final decision and by a *DataStore* to send its vote. Then, the *Item.Java* class represents the data stored in the *DataStore*. The class contains four different attributes:

  - value
    `type`: *int*
    Value of the Item.
  - version
    `type`: *int*
    Version of the Item.
  - locker
    `type`: *UUID*
    Lock of the Item.
  - valueChanged
    `type`: *boolean*
    Boolean flag to check if the value of the Item has changed after creation.

- **`message`**
  All types of messages exchanged among actors have been defined in this package. Each message implements the *Serialazible* interface and has the attribute *serialVersionUID*. Each message also overrides the method *toString* returning a Java *Gson* instance. There is an abstract class called *Message.Java* which includes the basic attribute for a message: the integer *sender id*. Then, there is a *StopMsg.Java* that defines a generic message to stop a client at any time. After that, there are other four packages:

  - snapshot
    Messages used to perform a global snapshot of the system have been defined here.
  - twopc
    Messages used to develop the two phase commit protocol have been defined in this package. There is a basic abstract message *TwoPcMessage.class* which includes the sender id, transaction id, and decision. The other messages extend it.
  - txn
    Messages involved in a transaction have been developed in this package. The basic

abstract message is *TxnMessage.Java* which extends the *Message.Java* class. Its attributes are the sender id and the transaction id. Other messages rely on it. There are also two packages called *read* and *write*. They contain the corresponding messages used for a read or write operation during a transaction.

  - welcome
    Messages exchanged at the start of the system have been defined here. There are three different welcome messages: *ClientWelcomeMessage.Java* to send to clients the coordinators references, *CoordinatorWelcomeMessage.Java* to send to coordinators the data stores references and *DataStoreWelcomeMessage.Java* to send to data stores the references of other data stores.

- `util`
  In this package there is a *JsonUtil.Java* class which defines the basic Gson builder to override the method *toString* for other classes. Also, there is a package *adapter.serialize* which contains a class *ActorRefSerializer*. It is used to serialize an instance of type *ActorRef* in order to print it correctly.

- `Main.Java`
  In this class, there is the public static method main where various actors are initialized and welcome messages are sent among them. Then, *run*(s) have been executed inside a loop considering the *MODE* defined in the *Configuration.Java* file.

- `Configuration.Java`
  In this class, there are all the possible configuration variables described in the chapter 2.

4 Protocol design

In this chapter we explain how we have designed our protocol to handle a transaction during all its phases following the 2PC protocol to keep the system consistent even during possible crashes.

4.1 Begin transaction

Each `Client` can perform a transaction formed by a random number of operations. The **minimum** number of operations is **20** and the **maximum** is **40**. These two constants can be modified inside the class *Client.java*. A `Client` can start a new transaction before sending to one of the possible `Coordinator` (chosen randomly) a `TxnBeginMessage` message. When the `Coordinator` receives the message, it associates a `UUID`[1] transaction identifier to that transaction. Then, it stores the transaction id and the client's *ActorRef* into a map called *transactionIdToClient*. Moreover, the `Coordinator` stores the `Client` id and the transaction id into another map called *clientIdToTransaction*. The project messages have been designed in a way that each message sent between the client and coordinator contains the client id and each message, regarding a transaction, between coordinator and data store contains the transaction id. Therefore, the map *transactionIdToClient* is used by the `Coordinator` to retrieve the `Client` *ActorRef* from a `Data Store` message and the map *clientIdToTransaction* is used by the `Coordinator` to retrieve the transaction id from a client message.

4.1.1 Private workspace

Every instance of a `Data Store` must keep track, inside a private workspace, of all operations (reads and/or writes) made on the affected *Item*(s) during a live *transaction*. Since a `Data Store` can handle multiple *transaction*(s) at a time internally uses a data structure, called *workspaces*, where all the operations associated to a transaction are stored. The *list* of operations is identified thanks to the transaction identifier (`UUID`) available on every message sent by the `Coordinator`. `workspaces` is of type: `Map<UUID, Map<Integer, Item»` The map's key is the transaction identifier and the value is a map/list of a one to one representation of the storage, where the key is the `Item`'s key (calculated using the actor's id) and the value is the `Item` object on which the operations have been made. Note that each `Item` in the workspace is a pure copy of the `Item` available in the storage. This has been to avoid conflicts and consistency problems across multiple transactions before the final `Coordinator`'s decision.
All changes present in a workspace are written to the "*persistent storage*" only if the current `Data Store` voted to *`COMMIT`* (during the voting phase) and the final decision taken by the `Coordinator` (after all votes have been received) is also to *`COMMIT`*. Otherwise, if the final decision of the `Coordinator` is to *`ABORT`*, the *persistent storage* is left unchanged, and the private workspace of the transaction is removed freeing useful memory. All the phases described before following the 2PC (2 Phase Commit) protocol to keep the system consistent.

4.2 End transaction

When the client ends operations, it sends a *TxnEndMessage* message with a decision to terminate the transaction. The client decision can be *`COMMIT`* or *`ABORT`*. When the coordinator receives it, the validation phase can start.

[1] A universally unique identifier (UUID) is a 128-bit label used for information in computer systems. [...] When generated according to the standard methods, UUIDs are, for practical purposes, unique. Their uniqueness does not depend on a central registration authority or coordination between the parties generating them, unlike most other numbering schemes. While the probability that a UUID will be duplicated is not zero, it is close enough to zero to be negligible[2].

4.2.1 Validation phase

The validation phase consists of developing the two phase commit algorithm to guarantee consistency among data stores. The project considers optimistic control to validate a transaction or not. The transaction will be committed or aborted considering the item versions and the locks on modified items acquired by a transaction. The two phase commit starts when the coordinator receives the *TxnEndMessage* from the client. Then, it retrieves the transaction id from the client id as it has been explained in the section 4.1. If the client wants to:

- ***COMMIT***: the coordinator retrieves data stores affected by that transaction. Then, it sends to all of them a *TwoPcVoteMessage* message. When a data store receives it, it checks if a transaction is valid or not calling the *canCommit* method. The `Data Store` checks the `Item` versions and locks to validate a transaction. First, it compares item versions in the private workspace with those in the storage. The check is passed if all item versions coincide. Then, the second check is passed if the data store is able to lock all items in storage that involved in the transaction. So, if both checks passed, the data store will send a *TwoPcVoteResultMessage* message to the coordinator voting ***COMMIT*** (YES). Otherwise, it will vote ***ABORT*** (NO). If at least one data store voted for ***ABORT*** (NO), the coordinator will immediately fix the final decision to ***ABORT*** and call the method *terminateTransaction*. If the coordinator is able to gather all votes, it means that all data stores voted for ***COMMIT*** (YES). Hence, the coordinator can fix the final decision to ***COMMIT*** and call the method *terminateTransaction*.

- ***ABORT***: the coordinator fixes the final decision to ***ABORT*** and calls the method *terminateTransaction*.

With the method *terminateTransaction* the coordinator informs affected data stores and the client of the final decision fixed before with *TwoPcDecisionMessage* message.

4.3 Crash and recovery

The system supports different type of crashes for both the `Coordinator` and the `Data Store`. Whenever an `Actor` crashes, after a fixed amount of time (`TWOPC_RECOVERY_TIMEOUT_MS`) it can recovery and return operative. The recovery is done thanks to a callback named `onTwoPcRecoveryMessage` where a message of type `TwoPcRecoveryMessage` is delivered to itself. Note that each `Actor` recovery procedure is different depending on the type, state, and context.

4.3.1 Crash

To crash an `Actor` we have made a utility method called `crash()` inside `Actor.java`. This method set a new `Receive` "*crashed*" state and spin up a timeout. The duration of the timeout is the time that the `Actor` is crashed. After the timeout is triggered, a `TwoPcRecoveryMessage` message is sent to itself starting the recovery phase.
All possible crash states are listed in the Configuration chapter 2. Each crash state starts with the word `CRASH_`. To sum up there are a total of 6 crash states.

4.3.2 Recovery

When the `Coordinator` recovers after a crash, firstly it changes its context to return "normal". Then, it fixes a decision for each pending transaction that affects a `Data Store`. If it has not decided to ***COMMIT*** or ***ABORT*** for a transaction before the crash, it will fix the final decision to ***ABORT***. Otherwise, the coordinator has already fixed a decision before the crash. After that, the coordinator calls the method *terminateTransaction* to communicate the final decision to affected data stores and clients. When the coordinator recovers after a crash, firstly it changes its context to return "normal". Then, it retrieves pending transactions from workspaces. If the data store has not voted yet for a transaction, it will safely ***ABORT*** unilaterally, sending to itself a *TwoPcDecisionMessage* message with final decision set to ***ABORT***. If the data store does not know the final decision, it will ask it directly to the coordinator that handled the transaction. Otherwise, it means that the data store has already known the final decision.

5 Implementation discussion

In this chapter we discuss problems, achievements, and improvements of the system.

5.1 Strict serializability

Banky, as written in section 1.4, follows the `Strict Serializability` model.
Informally, strict serializability means that operations appear to have occurred in some order, consistent with the real-time ordering of those operations; e.g. if operation A completes before operation B begins, then A should appear to precede B in the serialization order. Strict serializability is a transactional model: operations (usually termed "transactions") can involve several primitive operations performed in order. Strict serializability guarantees that operations take place atomically: a transaction's sub-operations do not appear to interleave with sub-operations from other transactions[4].
Thanks to the private workspaces mechanism, two transactions can read/modify the same `Item` asynchronously without interfering at each other. However, this freedom in the management of various operations by a transaction is valid until the `TxnEndMessage` is sent by the corresponding `Client`. The termination message involves a series of consistency checks and locking mechanisms on all `Item`(s) that are present in each single private workspace managed by one or multiple `Data Store`(s). As written in the previous chapters, if even a single check fails, the entire transaction and all the correlated operations done in the workspace are removed/ignored leaving the storage unmodified.
If two transactions have a common *Item*in their respective private workspace and want to *COMMIT* at the same time there must be a mechanism to ensure that the first transaction, managed at a hypothetical time `t0`, must certainly be allowed to `COMMIT` the changes before the second transaction arrive and is managed at the time `t1` where `t1` is greater than `t0`. Note that the previous statement takes for granted that on the 2PC vote request phase of the first transaction all `Data Store`(s) responded with `COMMIT` allowing the transaction to succeed.
Strict Serializability can also be seen as a distributed FIFO (First In, First Out) where the transactions that want to terminate are managed as a virtual queue. The first in the queue will be the first that starts the 2PC procedure and the last will be the last; and so on when a new transaction is added to the queue.
In addition to the position in the queue, we should also take in consideration that in the project there are random delays (can be turned off) that could also affect the locking and consistency checks done by one or more `Data Store`(s).
Note that a transaction that affects an `Item` that is also affected by another transaction and the first transaction is served first, can ***`ABORT`*** whether the second one can ***`COMMIT`***. The "*final decision*" on a transaction's status given only the order in the virtual queue is not deterministic.

5.2 Deadlock

There is a well-known blocking state since the validation phase is performed by two-phase commit. If the `Coordinator` crashes during the validation phase, `Data Store`(s) will perform the termination protocol. However, if no `Data Store`(s) knows the final decision, they are blocked. So, they can only wait for the `Coordinator` recovery. If the `Cooordinator` crashes, the `Data Store` handles it in the method called *onTwoPcTimeoutMessage*. Indeed, if the `Data Store` has voted for ***`COMMIT`*** and it does not know the final decision, it asks around to other `Data Store`(s). However, if no one knows the decision, the validation phase is blocked until the `Coordinator` recovery.

5.3 Stateful and memory

As other stateful architectures, our system suffers from high memory usage when multiple transactions are alive at the same time. Each `Coordinator` and `Data Store` have built in specialized structures to "*remember*" the state of a transaction. Even though a part of these resources is freed when the transaction end (commit or abort) there is a possibility that multiple `Client`(s) start a transaction and make a single operation (even a simple read) on a consistent set of `Item`(s) without terminating the transaction keeping the resources reserved. In a short time, the available resources are fully occupied causing real crashes and the impossibility of legitimate `Client`(s) to start or end a transaction.
This is a famous type of Denial-of-service attack.

5.4 Improvements

Regarding the section 5.2, one improvement could be to exploit the three-phase commit to perform the validation phase. In this way, there is no more the blocking state described previously, because if all `Data Store`(s) are in ready state they can safely ***`ABORT`***. Otherwise, if at least one `Data Store` is in ***`PRECOMMIT`*** state, they can safely ***`COMMIT`***. Briefly, the ***`READY`*** state is no more an uncertainty state. However, the three-phase commit introduces new possible blocking cases. Indeed, it does not tolerate network failures and total crash failures. This is because there is no protocol that guarantees correctness and non-blocking property.

Bibliography

[1] Wikipedia contributors. Log4j. https://wikipedia.org/wiki/Log4j.

[2] Wikipedia contributors. Universally unique identifier. https://wikipedia.org/wiki/Universally_unique_identifier.

[3] The Apache Software Foundation. Apache Log4j 2. https://logging.apache.org/log4j/2.x.

[4] LLC Jepsen. Strict Serializability. https://jepsen.io/consistency/models/strict-serializable.