Assignment 2 - Tuple Chat

Binghui Liu - 465580
Sergio A. Figueroa - 464439

December 18, 2014

1 General Ideas

The following scheme was tested to be correct against the unit tests provided and several user tests against the GUI (the latter performed mostly to test the functionality of the solution).

Every chat message will have an identifier composed by to fields of the tuple: the channel id and a consecutive number. The consecutive number allows to keep track of the order of the messages, and will be used, in addition to a counter of how many times it has been read, to determine if a message has been read by all the listeners. If the message has been read and is outside the size of the buffer, the message will be deleted. A reader should not read the same message twice.

These decisions will be explained technically in the next sections.

2 TupleSpace

The behaviour of the tuple space is very simple. The only requirement that should be expected is consistency:

- If `put` is executed, the tuple must be correctly stored.
- If `read` is executed, a tuple matching the pattern provided should be returned.
- If `get` is executed, a tuple matching the pattern provided should be returned **and removed** from the tuple space. The same tuple should not be returned as the result of more than one call to `get`.

In other words, from the concurrency point of view, it is necessary to guarantee the atomicity of each operation. In addition, since the tuple space is likely to be intensive in writing and reading, a time efficient implementation is desirable.

2.1 Basic data structures

This section presents the data types and structures that will define the tuple space:

- **Tuple:** `String[]`

- **TupleSpace:** `HashMap<Integer, LinkedList<Tuple>>`

An additional comment about the data structure chosen to contain the tuple space is in order. The hash map will allow to immediately filter tuples that don't match the length of the pattern. Once those are filtered, every element on the linked list will be compared again the pattern, and the matching elements will be returned.

2.2 Searching

To allow the execution of several non-related queries, the tuple space hash map will not be synchronized during a searching operation. On the other hand, the linked list containing all the tuples of the size of the pattern will be obtained and locked. But if there is not a tuple that matches the pattern, an empty `LinkedList` will be added to the `HashMap` for the corresponding key. The synchronization on the list prevents inconsistent concurrent readings. If there is no matching result, the get call will wait until that situation changes.

2.3 Tuple space operations

Both `get` and `read` are searching operations. Then only difference between them is that the former, after the search, deletes the obtained tuple from the tuple space.

The execution of `put`, on the other hand, requires two kinds of locks. First, the `HashMap` is synchronized. If there is already an entry for tuples of that given tuple length, the tuple will be added to the existing `LinkedList`. If not, it will be necessary to create a new entry `<length, LinkedList>` containing only the tuple being put. In both cases, the resulting `LinkedList` will acquire a lock to prevent dirty readings.

3 Tuples and structures (ChatServer and ChatListener)

The logic of the chat works mostly on a peer to peer basis, in the sense that there is no central server handling the channels or the incoming communications. There is, however, a central point that regulates the flow of all the communication: a tuple space server. Every running instance of the chat will contain a connection to a tuple space server. This section describes the different kind of tuples that can populate the tuple space in order to allow the chat server to work.

3.1 Status check

Structure: `("STATUS", String:[channel1:rows, channel2:rows, , channeln:rows])`

This tuple will be put by the *master* peer (i.e. the first one to connect) and should not be removed (implying also no modification). It contains the amount of rows that should be buffered and a series of strings containing the names of the channels.

3.2 Messages

Structure: `(String:channel name, ''MESSAGE'', int:listeners, int:position, String:message)`

Every message tuple contains the name of the channel (`channel_name`) used, a consecutive index (`position`) and the content of the `message`. Also, every channel is a *flexible* bounded buffer: it has a maximum capacity (`rows`, defined on the `STATUS` tuple), but this capacity can be exceeded when there is a slow reader still waiting to read. For this reason, it is important to keep track of the amount of readers subscribed to the channel and on the amount of subscribers that have not read a certain message. The tuple contains also the index of the message in that channel.

It is important to emphasize: a message tuple shall **only** be deleted when it is necessary to make room for a new message in the buffer (i.e. **the buffer is full**) and **all the listeners have consumed it**.

3.3 Write lock

Structure: `(String:channel_name, ''WRITE'', int:position)`

There is a `WRITE` tuple for each channel. It is used as the mechanism to prevent concurrent dirty writings, as only one server should be able to `get` it at a given time. The variable `position` contains the value of the next available message index (defined as a consecutive).

The tuple is created by the first server and should be retrieved only by `get`. The server returns the tuple after writing the message, incrementing the value of `position`.

3.4 Read buffer

Structure: `(String:channel_name, ''READ'', int:min_position, int:max_position)`

The maximum size of the buffer is defined by the field `rows` of every channel. The actual state of the buffer, expressed in terms of the position where it starts and the position where it ends, is stored in this tuple. At no point should occur that `max_position - min_position > rows`.

Note that there could still be messages with an index lower than `min_position`, if they have not been read by all the registered listeners.

3.5 Connected listeners count

Structure: `(String:channel_name, ''CONN'', int:listeners)`

The tuple keeps track of the amount of listeners connected to a new channel. Every time a new listener connects, the value of `listeners` is incremented by 1. Every time it disconnects, the value is decreased by one.

The tuple is created by the first server, updated by the arriving server, and is consumed by the means of `read`.

3.6 Outside of the tuple space

The chat server should also maintain a `HashMap` of channels in order to keep record of how many elements there are in every channel.Every `ChatListener` has a variable `latest_postion` to record which position it should read from its channel buffer, which should be initialized at the begining of the connection.

4 Concurrency comments

As in the previous assignment, the concurrency of the system relies heavily on the implementation of the tuple space. If the atomicity of the tuple space is implemented properly, the tuples can be used to provide concurrency features.

However, in this scenario the implementation of the chat needs to be careful in order to prevent deadlocks. For instance: if the chat calls get instead of `read` on the status tuple, every further peer attempting to connect would deadlock.

5 Implementation details

There were several implementation issues that were troubleshooted by a combination of standard Java debugging, JUnit test result analysis and retrieving the status of the a given tuple to understand the behavior.

The most relevant issue presented when connecting the server and the listener. First, it was only possible to create one listener per channel, since the next locked. Then, it was possible to create the listeners but then the chat did not keep history. Finally, the history was retrieved but the new messages only arrived to the first listener.

All the issues were generated by two mishappenings: first, some tuples were retrieved as locks but not put back again because the process was waiting for yet another tuple (deadlocking). Second of them, the counters implemented to keep track of the current buffer position were improperly initialized.

A way to analyze the whole content of the TupleSpace at a given moment, or to handle it (cleaning it, specially), would have been very useful for the development of this assignment.