Workbook 5

Introduction

Last week you wrote your own implementation of a Java chat server. In a chat system, it is desirable to preserve the causal ordering of messages displayed in the client. For example, if message M is written on client A in response to a message L from client B, then we should ensure that L is displayed before M on all clients.

Your implementation of a Java chat server currently ensures causal ordering by passing all messages through a single `MultiQueue`, however this does not scale for a large system. A more scaleable solution distributes message delivery across multiple processors, or even multiple servers; however this may result in message reordering. To fix this, clients need to reorder messages in order to guarantee messages are displayed in causal order. Therefore, this week, you will add support for vector clocks to your chat client in order to ensure message causality is preserved in this setting.

Important

An on-line version of this guide is available at:

http://www.cl.cam.ac.uk/teaching/current/FJava

You should check this page regularly for announcements and errata. You might find it useful to refer to the on-line version of this guide in order to follow any provided web links or to cut 'n' paste example code.

Vector clocks

A vector clock is a data structure and associated algorithm for determining the partial order of events in a distributed system. In other words, the algorithm allows clients to determine whether, given two events D and E, if E occurred after D, D occurred after E, or whether E and D were concurrent. The algorithm requires each client to maintain a vector of counters, called a *vector clock*. The client's current vector clock is attached to events when they happen, and the vector clocks attached to events can subsequently be compared to determine the partial order of events.

In text books, the vector clock is often presented as a mathematical vector of integers; this works fine for a closed system where the number of communicating processes are known when the system is designed. However, in our chat system, we do not know in advance how many clients there will be. Therefore in this workbook we will represent the vector clock by `java.util.Map<String, Integer>` where each key in the map represents a client which should be represented by a random unique identifier generated with the `java.util.UUID` class. The value in the `Map` should be an integer fulfilling the criteria for the vector clock algorithm, as explained below.

We use the notation `clock[X]` to access the integer, or *clock element*, associated with the unique identifier `X` in a vector clock called `clock`; and the notation `clock[X] += 1` to increment the integer associated with the unique identifier `X` by one.

Updating the vector clock

Vector clocks are typically used to track events in distributed systems, and as part of the protocol a client will increment its own clock element on any event, including when a local event occurs or when it sends (or receives) a message from (to) another client. This is because communication is, after all, just an event too. In the messaging context, we want to track the number of messages sent by each client and therefore require a client to only increment its own clock element whenever it sends a message; it does *not* increment its own clock element when receiving a message. This change makes the task of reconstructing the order of messages for display possible later in this workbook. This algorithm is presented in the Part IB Concurrent and Distributed Systems course under the name of 'causal broadcast'.

The rules for updating vector clocks are therefore as follows:

- Each chat client should maintain its own local vector clock for as long as the client is alive.
- When a client with identifier `ID`, and a vector clock called `client` sends a message, the client must increment its own clock element by one (i.e. `client[ID] += 1`) before attaching a copy of its own vector clock to the message.
- When a client with identifier `ID` receives a message, it updates the clock elements in its own vector clock to be the larger of those in its own clock (called `client`) and the values found in the message (called `msg`). In other words, for all identifiers `U`, `client[U] = max(client[U], msg[U])`; if there are identifiers found only in the `client` or `msg` clocks, then those identifiers and values should be included in the new `client` vector clock.

Happened-before relationship

Given the vector clocks attached to two messages, the vector clocks can be used to determine whether message L happened before message M, M happened before L, or whether M and L are concurrent. Formally, L happened before M iff all the clock elements in L are less than or equal to all clock elements in M and at least one element is strictly smaller in L than in M. If one clock has elements not present in the other, then you should assume the missing clock element has the value zero. If neither L happened before M, nor M happened before L, then M and L are concurrent.

You are now ready to produce your own implementation of the vector clock algorithm.

1. Start by visiting the Ticklet 5 project on Chime https://www.cl.cam.ac.uk/teaching/current/FJava/ticklet5 and cloning a copy of your repository onto your local machine.
2. Take a look at `uk.ac.cam.cl.fjava.messages.Message`. Can you see where the vector clock has been introduced into the message format?
3. Complete the implementation of `VectorClock` by filling in the sections marked `TODO`.

Reorder buffer

Your next challenge is to display messages in the correct order. If two messages are causally dependent on each other then the chat client *must* display the earlier message first. Otherwise, if two messages are not causally dependent on each other, the chat client *must* display the messages in the same order that they were sent by the server. You are welcome to design your ordering algorithm as you wish, provided that it meets these two criteria.

One simple and effective approach to meet these criteria is to construct a buffer of messages together with a `VectorClock` called `lastDisplayed` to represent the last message displayed by the client. Initially, the buffer will be empty. When the *first* message arrives the value of `lastDisplayed` should be set equal to the vector clock of the first message. Any subsequent messages which arrive and happened before this first message should be dropped; this ensures that the display does not show older messages out-of-order.

Whenever a new message arrives, it should be placed at the end of the buffer. Then, scan the buffer from the beginning, looking for a message, M, where all but one of the values in the vector clock are less than or equal to those found in `lastDisplayed` and the remaining value in M is exactly one greater; if such a message is found, this message can be displayed. If such a message M has been found with a vector clock called `msg`, the value of `lastDisplayed` should be updated with `lastDisplayed.updateClock(msg)`.

Since the arrival of a single message may result in several messages becoming eligible for display, the buffer should be repeatedly scanned from the beginning every time a new message is displayed until either the buffer is empty or there are no more messages eligible for display.

4. Complete your implementation of `ReorderBuffer`. You can use the algorithm described above, or your own, provided it meets the two criteria described.

Chat client

Your final task is to upgrade your Chat client to make use of your implementation of `VectorClock` and `ReorderBuffer`. To do so you should start by copying your implementation of the `ChatClient` from Ticklet 2 and then updating it to ensure:

- The client generates a random unique identifier using the `java.util.UUID` class; the result will be of the form "bc79adc3-985f-4ac4-8e6f-01344dafb963". The unique identifier should be converted to a `String` kept for the entire execution time of the program.
- The client should have a `VectorClock` object.
- Whenever any message is received, the client's vector clock should be updated correctly, the message added to the reorder buffer, and any messages which can now be displayed are shown to the user.
- Whenever any message is sent, a copy of the client's vector clock is updated and included in the message sent to the server. NB: The clock element associated with the client's own unique identifier in the vector clock attached to the first message should be equal to 1; in the second message, 2; and so on. The need to include vector clocks in all messages requires you to make use of the new constructors for *all* the classes found in the package `uk.ac.cam.cl.fjava.messages`.

5. Complete your implementation of `ChatClient` as specified above.
6. Test your implementation of your `ChatClient` by connecting to port 15007 on `java-1b.cl.cam.ac.uk`. In particular, your client will be sent a set of messages which have been carefully crafted to allow you to check that you display the messages in the correct order. If the messages are not displayed in incrementing order in your client, then you have a bug; please fix it before submitting to the automated tester as we retest your code in the same way.
7. Once you are sure your implementation of your `ChatClient` works on the set of test messages, you may connect to port 15006 on `java-1b.cl.cam.ac.uk` to access the new version of the chat server which has support for vector clocks.
8. Can you successfully connect the latest implementation of your `ChatClient` to the old version of the chat server on port 15004? Why? Be prepared to discuss this with your Ticker next week.

Ticklet 5

You have now completed all the necessary code to gain your fifth ticklet. Your repository should contain the following source files:

```
src/uk/ac/cam/crsid/fjava/tick5/ChatClient.java
src/uk/ac/cam/crsid/fjava/tick5/ReorderBuffer.java
src/uk/ac/cam/crsid/fjava/tick5/VectorClock.java
src/uk/ac/cam/cl/fjava/messages/ChangeNickMessage.java
src/uk/ac/cam/cl/fjava/messages/ChatMessage.java
src/uk/ac/cam/cl/fjava/messages/NewMessageType.java
src/uk/ac/cam/cl/fjava/messages/Message.java
src/uk/ac/cam/cl/fjava/messages/RelayMessage.java
src/uk/ac/cam/cl/fjava/messages/StatusMessage.java
```

When you are satisfied you have completed everything, you should commit all outstanding changes and push these to the Chime server. On the Chime server, check that the latest version of your files are in the repository, and once you are happy schedule your code for testing. You can resubmit as many times as you like and there is no penalty for re-submission. If, after waiting one hour, you have not received a final response you should notify `ticks1b-admin@cl.cam.ac.uk` of the problem. You should submit a version of your code which successfully passes the automated checks by the deadline, so don't leave it to the last minute!