Distributed Systems Lab 12

Eventual consistency

During the previous labs, we presented Raft, a distributed consensus algorithm, which can be used to guarantee sequential consistency across all processes of a distributed system. This lab discusses how a weaker form of consistency, **eventual consistency**, can be implemented in a distributed system, employing as a running example a distributed text editor. We will use files contained in this package, which you should download and extract locally.

Learning section

Eventual consistency

Ensuring sequential consistency across a distributed system boils down to employing some consensus algorithm, which usually involves introducing a significant overhead on the solution. However, not all systems require such a strong consistency requirement to operate correctly. For instance, consider a distributed text editor, which enables users to independently and simultaneously edit some shared text. Usually, it is not required that at any time all users see exactly the same text. On the contrary, edits can be displayed to different users in various orders. It is only required that all users will be displayed the same text when they all stop typing. In other words, the distributed text editor shall guarantee **eventual consistency**.

There are two approaches to state synchronization between processes of such a system. In the **state-based** approach processes exchange their local states, converging eventually to the same final state across the system. In the **op-based** approach all processes start with the same local state and then, by exchanging operations to execute, they eventually converge to the same final state too. During this lab, we will discuss the latter approach, employing the distributed text editor as a running example.

Op-based distributed text editor: Idea

In the considered distributed text editor, each process stores locally a copy of the edited text. All processes start with the same initial text and then they independently update their texts by applying operations issued by themselves and by other processes. We will consider two types of operations:

1. inserting a character at a position:

```
insert(position, character, issuing process's rank)
```

2. deleting a character at a position:

```
delete(position, issuing process's rank)
```

Operations are issued with respect to the current text of the issuing process.

From a global perspective, operations in the system constitute a partially ordered set: operation *A* is *newer* than operation *B* if some process issued *A* after it had applied *B* to its text (*A has already seen effects of B*); operation *A* is *older* than operation *B* if some process had issued *A* before it issued *B* (*B has already seen the effects of A*); operation *A* is *concurrent* with operation *B* if it is neither newer nor older than *B* (*A has not seen the effects of B, B has not seen the effects of A* as can happen when two processes issue these operations concurrently).

When some process issues an operation, it applies the operation to its text and broadcasts the operation to the other processes. The processes, however, cannot just apply the operation as they receive it: the operations as defined previously are not commutative and thus applying them in an arbitrary order would not guarantee convergence to the same texts. Therefore, at least two following conditions have to be met.

First, the received operation has to be applied only when all operations older than it have already been applied, as otherwise the partial order would not be preserved. If some older operations have not been already received and applied by the process, the process shall store the operation aside and apply it later when it is possible.

Second, the operation has to be transformed with respect to the already applied operations it is concurrent with. The goal of the **transformations** is to exploit commutativity: if two operations are concurrent, different processes may execute them in different orders, but to reach eventual consistency, these executions have to yield the same texts nevertheless. This can be achieved by transformation rules defined, for instance, as follows:

```
1 Transform insert(p1, c1, r1) wrt. insert(p2, c2, r2) :-
2   if p1 < p2: insert(p1, c1, r1)
3   if p1 = p2 and r1 < r2: insert(p1, c1, r1)
4   else: insert(p1 + 1, c1, r1)
```

Transformation rules for the remaining combinations of operations can be defined analogously. This way, when there are two concurrent operations, *A* and *B*, in the system, applying first *A* and then *B* transformed with respect to *A* yields the same eventual text as applying first *B* and then *A* transformed with respect to *B*.

To facilitate discovering which transformations have to be applied to an operation, each process stores locally a log of operations it has already applied to its text. When a process receives a new operation issued by another process and it has already applied all operations older than this one, it examines its log (the least recent operation first) and transforms the received operation with respect to all operations from its log that are concurrent with the operation. Then, it appends the transformed operation to the log and applies it to the text.

A naive approach to discovering which operations are older than and which are concurrent with a new operation would be to send and store with each operation a log of operations applied before this one. However, the problem can be solved in a much more performant way by using **vector clocks**.

Vector clock

Vector clocks were discussed at length during lectures. As a reminder, a vector clock is an array-like structure that stores independent counters. A vector clock can be used by a process of the distributed text editor to track operations applied to the text: the *i*-th counter of the vector clock counts applied operations issued by the *i*-th process of the system (i.e., the process with rank *i*). After a process applies an operation (issued by itself or by another process), it increments the corresponding counter in its vector clock. The operation is appended to the log together with the current state of the vector clock. When the process broadcasts the operation, the associated vector clock state is attached. This way, a process that receives the operation can compare this vector clock with the vector clocks of operations stored in its log:

- An operation *A* is considered older than or equal to an operation *B* if all counters of *A*'s vector clock are smaller than or equal to the corresponding counters of *B*'s vector clock.
- If neither operation is considered older than or equal to the other, *A* and *B* are considered concurrent.

Vector clocks are also used to verify if a process has already applied all operations older than the received one: the process's current vector clock has to have all counters equal to or greater than the respective counters of the received operation's vector clock, except for the counter corresponding to the issuing process: it has to be smaller precisely by one in the process's vector clock.

Broadcast guarantees

The overall correctness of the op-based eventual consistency algorithm depends also on guarantees provided by the employed broadcast primitive and may require introducing into the algorithm additional features depending on how the actual message delivery performs.

For instance, if operations are not guaranteed to be delivered in the order they are sent, processes should store aside operations which cannot be applied already when they are received and should try to apply them later. If operations are not guaranteed to be delivered to each process exactly once, processes should first verify that they have not already applied them (unless operations are idempotent), and should implement some retransmission mechanisms if message delivery can fail. And so forth.

Example

An example activity of an op-based distributed text editor is illustrated in a diagram. The system consists of two processes with ranks `0` and 1. Both processes start with empty texts, zeroed vector clocks, and empty logs.

First, the process with rank `0` issues the `insert(0, 'i', 0)` operation, applies it to its text, increments its vector clock, appends the operation to its log, and broadcasts the operation with the associated vector clock to the other process. The other process has an empty log, so it just applies the operation to its text as is, increments its vector clock, and appends the operation to its log. Both process store now texts `"i"`.

Then, the process with rank `0` issues the `insert(0, 'H', 0)` operation, applies the operation to its text (obtaining `"Hi"`), increments its vector clock, appends the operation to its log, and broadcast the operation to the other process. Concurrently, the process with rank 1 issues the `insert(1, '!', 1)` operation, applies the operation to its text (obtaining `"i!"`), increments its vector clock, appends the operation to its log, and broadcasts the operation to the other process.

When the process with rank `0` receives the `insert(1, '!', 1) [1, 1]` operation, it examines its log by comparing the received vector clock with vector clocks of operations stored in the log: the new operation is newer than the first stored operation (so no transformation is needed), but concurrent with the second stored operation (so a transformation is needed). It transforms the new operation with respect to the second stored operation and obtains `insert(2, '!', 1)`. The process applies the transformed operation to its text, increments its vector clock, and appends the operation to its log.

By symmetry, when the process with rank 1 receives the `insert(0, 'H', 0) [2, 0]` operation, it examines its log by comparing the received vector clock with vector clocks of operations stored in the log: the new operation is newer than the first stored operation (so no transformation is needed), but concurrent with the second stored operation (so a transformation is needed). It transforms the new operation with respect to the second stored operation and obtains `insert(0, 'H', 0)`. The process applies the transformed operation to its text, increments its vector clock, and appends the operation to its log.

Eventually, both processes obtain the same text: `"Hi!"`.

Op-based distributed text editor: Disclaimer

The above description, however, illustrates only an idea how eventual consistency can be achieved in a distributed text editor: the presented algorithm includes a few shortcomings. For example, it does not handle situations when there are two causally-related operations that are both concurrent with a third operation.

Nevertheless, the algorithm can be extended to truly guarantee eventual consistency by examining the log and applying transformations in a more sophisticated way, depending on additional guarantees one wants to provide (see, for instance, this paper).

Small Assignment

Your task is to implement a distributed text editor. However, given that an algorithm that guarantees truly eventual consistency is too intricate for a Small Assignment, you are to implement the algorithm presented hitherto but modified so as to guarantee text convergence through process synchronization.

Algorithm for Small Assignment

Processes operate in implicit rounds. At the beginning of a new round, each process issues its operation (with respect to its current text, that is, the text resulting from the previous round), appends the operation to the log, applies it to the text, and broadcasts to other processes. Then, it waits for operations from other processes and handles them (i.e., transforms, appends to the log, and applies them to the text). The round ends when the process receives operations from all other processes. When the process receives an operation from the next round (every process manages the rounds independently, so some may already start the next round and broadcast its next operation), the operation is not processed until the next round. Operations within a round are always processed in the order they are received.

When at the beginning of a round a process does not have an operation to be issued, it waits for the next event. If the event is a user's request of a new edit, then the process continues with this operation. If it is a reception of some other process's operation, the process internally issues itself a `NOP` operation, handles it, and then continues processing the received operation. Effectively, an operation issued by the process itself is always the first operation appended to its log in each round.

This way, in every round, each process issues exactly one operation. The operation is newer than all operations issued in the previous rounds and concurrent with all operations issued by other processes in the same round. Therefore, there is no need to use vector clocks. Moreover, all concurrent operations are issued with respect to the same text.

The modified algorithm is illustrated in a diagram.

Implementation specification

The system consists of processes, clients, and a broadcast module.

The processes manage the exchange and transformations of text operations. There is a constant number of processes in the system, and they are assigned consecutive ranks (starting with rank `0`).

You can assume the processes do not fail.

Operation transformations shall be implemented as follows:

```
Transform insert(p1, c1, r1) wrt. insert(p2, c2, r2) :-
  if p1 < p2: insert(p1, c1, r1)
  if p1 = p2 and r1 < r2: insert(p1, c1, r1)
  else: insert(p1 + 1, c1, r1)

Transform delete(p1, r1) wrt. delete(p2, r2) :-
  if p1 < p2: delete(p1, r1)
  if p1 = p2: NOP (do not modify text)
  else: delete(p1 - 1, r1)

Transform insert(p1, c1, r1) wrt. delete(p2, r2) :-
  if p1 ≤ p2: insert(p1, c1, r1)
  else: insert(p1 - 1, c1, r1)

Transform delete(p1, r1) wrt. insert(p2, c2, r2) :-
  if p1 < p2: delete(p1, r1)
  else: delete(p1 + 1, r1)
```

In contrast to the text editor described above, the processes do not store texts or issue commands themselves. In the assignment, these activities are done by the clients.

Each client is coupled with exactly one process. To issue an operation, the client sends an `EditRequest` message to its process. It can issue multiple edit requests at once. You can assume that the requests are valid edits (with respect to the text as of they are issued). The process shall process requests in the order they are received, introducing them to the system one by one in the first possible rounds. The process sends to its client an `Edit` message each time the client should apply a new operation to its text (including operations issued by the client itself and the `NOP` operation). Since the client communicates with its process in an asynchronous way, in each edit request, the client sends also the total number of operations it has applied to its text so far. This way, the process can learn with respect to which state the operation was issued, and transforms the request with respect to operations which were appended to the log later. These transformations shall follow the same transformation rules as concurrent operations, and the edit request shall be assigned a temporary process rank larger than ranks of actual processes (e.g., `N+1` in a system containing `N` processes). After the transformations, the requested operation is appended to the log, broadcast, and sent back to the client (which applies it to the text only then).

To broadcast an operation to other processes, the process shall send an `Operation` message to the broadcast module. Each message will be eventually delivered to every other process exactly once, and messages from a process will be delivered in the order they were sent by this process. You are not allowed to send messages between processes other than through the broadcast module.

You shall complete the implementation of the process and necessary data structures provided in the template, following the interfaces and doc comments defined there. Exemplary implementations of the client and the broadcast module are already provided.

This Small Assignment is worth **2 points**. To run the system you should use the executor system you implemented as the first Large Assignment.

Additional Homework

To learn how to implement true eventual consistency in an op-based distributed text editor we recommend reading the aforementioned paper by M. Ressel et al.

Authors: K. Iwanicki, M. Banaszek, W. Ciszewski, M. Matraszek.