

CS138 Programming Assignment 3: Virtual Synchrony

<i>Assignment Out:</i>	Mar. 16, 2009
<i>Helpsession:</i>	Mar. 17, 2009 (Location TBD, 7pm)
<i>Assignment Due:</i>	Apr. 8, 2009 (11:59 pm)

1 Virtual Synchrony

Many distributed applications require guarantees about the order in which updates to replicated data are applied in an environment where many clients may be mutating data simultaneously. For these applications, it is usually not enough to simply see updates in the same order at different replicas, however. Stronger restrictions may require that clients' mutations remain in the same order that the client makes them. An even stronger restriction is that mutations be ordered in a causally consistent way. Causal ordering ensures that if a client decides to update some data based on the current value of some other replicated data, the mutation that caused the referenced data to have its current value will be applied before the subsequent update, regardless of whether two different clients made those updates. Finally, the strongest guarantee that may be made applies mutations in a causally and totally ordered way - all mutations are applied in the same causal order at all replicas.

In this project, you will be implementing the CBCAST and ABCAST protocols. The CBCAST protocol supports causally ordered messaging between members of a distributed process group. The ABCAST protocol supports totally ordered message delivery on top of CBCAST. Your implementation will allow nodes to join and leave a process group without interrupting service from the client application's point of view.

1.1 Process Group Communication

For the CBCAST and ABCAST protocols, a set of distributed nodes that replicate data in the same dataset are known as a process group. Communication within process groups is preformed by **multicasting** messages. When a process multicasts a message, it sends a copy of the message to each node in the process group over a FIFO communication channel. This guarantees that the messages sent by one process to another will arrive at the destination in the order that they were sent. Every node (including the sending node) receives the same message. Messages usually contain an updated value for a key-value pair and may also carry other data required by the protocol. A message is received by a process and then *delayed* until the conditions for its *delivery* are met. The necessary conditions for delivery will be developed in the next few sections.

1.2 Logical Time

To deliver messages in a causal order, a notion of the time at which a message was sent is required. Synchronizing real-time clocks across multiple machines is practically impossible, and even if it were possible, a real time clock might not be able to provide the granularity needed to accurately order events that often take place over a few processor cycles. For this reason, the notion of logical clocks has been developed. In a CBCAST process group, logical clocks take the form of vectors of integers with entries for each process in the group. These vectors are known as **vector clocks**. The integer entries measure the number of messages that have been multicasted by a process. Each process p_i maintains its own vector clock $VC(p_i)$, and every time the process multicasts a message, it increments its own entry $VC(p_i)[i]$ in that vector clock. A copy of the vector clock at the time the message was multicasted is included with the message.

Whenever a message is “delivered” by a process, that process merges the vector clock that was attached to the message with its own. Merging is performed by creating a new vector clock with each entry equal to the largest value in the corresponding entries of the two vector clocks. By incrementing its local entry in the vector clock as outgoing messages are sent and merging clocks when incoming messages are delivered, a process can represent precisely the number of messages from each member of the process group that the process is aware of.

1.3 Causal Ordering

To maintain causal ordering, a process p_i must delay the delivery of a multicasted message m that it has received from process p_j until

$$\text{for all entries } k \neq j, VC(m)[k] \leq VC(p_i)[k]$$

and

$$VC(m)[j] = VC(p_i)[j] + 1.$$

The first condition ensures that m is only delivered at p_i once all messages that were delivered at p_j have also been delivered at p_i . The second condition ensures that m is the oldest message sent from p_j that p_i has not yet delivered. Together, these conditions are sufficient to maintain causal ordering. The processes in a CBCAST process group use these delivery rules to deliver multicast messages.

1.4 Total Ordering

The ABCAST protocol is designed to provide total ordering in addition to causal ordering. To implement the protocol, the processes in a CBCAST process group must agree to deliver causally simultaneous messages in the same order. For instance, in a process group made up of 3 processes each of which have sent at least 1 multicast message, two messages carrying the vector clocks

$$VC(m_1) = [2 \ 1 \ 1] \text{ and } VC(m_2) = [1 \ 1 \ 2]$$

are causally independent and therefore it is possible to deliver m_1 before m_2 or m_2 before m_1 .

Because it is possible to place a total order on some messages while continuing to provide only causal ordering on others, we need to distinguish between the types of messages. Processes may both CBCAST and ABCAST messages, and the ABCAST messages are marked as such. To provide a total order on causally independent messages, we begin by assigning one member of a process group to be the **coordinator**. The coordinator delivers ABCAST messages with the same rules for which it delivers CBCAST messages, but it also keeps track of the unique identifiers of the ABCAST messages that it delivers and the order in which it has delivered them. Periodically, the coordinator CBCASTs a **sets-order** message containing the UIDs of the ABCAST messages it has delivered and their order. When process p_i receives the sets-order message, it immediately makes note of the order that the message carries and attempts to deliver any ABCAST messages that it has delayed. An ABCAST message may only be delivered if the criteria is met to deliver it as a CBCAST message and it is the next message in the list specified by the sets-order message. Eventually, all the ABCAST messages identified in the sets-order message will be received by p_i , and p_i will be able to deliver each delayed ABCAST message, in addition to the sets-order message itself. Delivering the sets-order message has no effect except to allow the delivery of subsequent CBCAST messages from the coordinator.

2 Dynamic Process Groups

The CBCAST and ABCAST protocols described in the previous section do not accommodate the addition or removal of processes from a process group. We can extend CBCAST to allow the membership of the process group to change over time by introducing the concept of “flushing” communication within a process group. Our extended protocol will allow one process to be added or removed from the process group at a time. To differentiate between the states of a process group as members are added and leave, we define “view” i of a process group to be the set of process in the group at time i . The views i and $i + 1$ of a particular process group may only differ by the addition or removal of one process.

2.1 Flushing

When a process p becomes aware of a new view (suppose that a node that would like to be added to the process group contacts p), p may CBCAST a **flush** message and delay all subsequent multicasts until the flush protocol has completed. On receipt of this message, the other processes know that they will not receive any more messages from p until the flush protocol has completed, since their communication channels are FIFO. Each process will multicast its own flush message as it learns of the new view. Although processes who have sent a flush message will no longer send new multicast messages, they will continue to receive and deliver messages. Once a process

has delivered flush messages from every process in the process group (including itself), the process considers the flush protocol to be complete.

At the completion of the flush protocol, a process makes any necessary changes to its internal state before multicasting and receiving messages in the new view. First, it should update its list of process group peers to reflect the membership of the new view. Because the number of processes in the group has changed, it is also necessary to reset the process' vector clock. Since all processes will have the same state after completing the flush protocol, they may zero the entries of their vector clocks and modify the size to match the number of processes in the new view. The operation of the process group can then continue normally.

2.2 Quorum

One final implementation detail is necessary to safely implement the flush protocol. If two or more different processes choose to join or leave the same process group simultaneously, we must ensure that at most one process is successful in its request. To this end, each process in the group is asked to vote on adding or removing a specific process from the group. If a process is asked to vote and it has not already voted in another quorum, it votes yes. Once a process has voted yes in one quorum, it must vote no in any subsequent quorums unless it is informed that the quorum it voted yes in failed. A quorum succeeds if strictly more than half of the nodes in the group vote yes. Once a quorum has succeeded, the process that initiated the quorum can authoritatively initiate the flush protocol at every process in the group. Once the flush protocol has completed and the new view is installed, processes are free to vote in another quorum for the new view.

3 The Assignment

You need to implement the following methods in `SynchronyNodeLocal.java`:

```
public boolean quorum(CachedNode<SynchronyNodePrivate> cache, boolean add, boolean reset)
private void process()
private boolean deliver(SynchronyMessage m)
```

The amount of code you must write is not large, but it is tricky. Because so much of this assignment is already implemented for you in the support code, we'd like you to answer a few simple questions about the code in your README in addition to filling in the methods we have left for you to write.

1. Why is there a single thread for delivering messages and single thread for sending outgoing messages? What property of communication in the CBCAST/ABCAST protocol does this help maintain?

2. Why do the sets-order portions of received messages get processed immediately, rather than when the message carrying the sets order is actually delivered?
3. Why is additional state (beyond the node's vector clock) required to determine which of the node's own multicast messages to deliver next?
4. In our particular implementation, why does the coordinator wait to multicast its flush message until it has received flush messages from all the other nodes?

4 Testing

The way in which you test your code for this project is incredibly important. Debugging and verifying the correctness of your implementation will be difficult. You must turn in test code that demonstrates that you have checked your implementation thoroughly. **A full 15% of your grade will be based on the quality of your test code.** Specifically, you should write test code that shows that updates to key-value pairs replicated by your `SynchronyNode` implementations are applied in exactly the same order at each node, even if nodes are joining the process group as updates are being made. The support code that you have been given already provides support for the `SynchronyNodeMonitor` interface, which is a good starting point for writing your test code.

Depending on your implementation, there may be ample room for race conditions that only take effect when the timing of sending and receiving messages is just right. **Adding artificial delays to the message passing code is an essential part of testing your implementation.** Try a number of distributions of random delays before handing in your code.

5 Extra Credit

You may implement fault-tolerance in this assignment for extra credit. To get an idea of what is involved, ask the TAs. Don't try this before finishing the rest of the assignment.

6 Code Exchange

To get started, copy the `/course/cs138/asgn/synchrony` directory to your CS138 course directory. Please note that this assignment makes use of the CS138 support code library, which should be added to the `CLASSPATH` of your IDE if you choose to use one. The `support.jar` library can be found in `/course/cs138/pub/lib/` and the source code for the support library is available in `/course/cs138/pub/src/support.zip`. The `ant` build file which you have been given already adds `support.jar` to your `CLASSPATH` for compiling, running, and debugging. Documentation for the support library is available at <http://cs.brown.edu/courses/doc/api/>.

7 Handing in

You need to write a README, documenting any bugs in your code, any extra features you added, and anything else you think the TAs should know about your project. Your README also needs to contain the answers to the questions posed earlier in the handout. You should hand in `synchrony` by running `"/course/cs138/bin/cs138_handin synchrony"` from your project's directory.