Design Document

A Doodle-like Poll Service with Joram

Anindita Das
dasanuiit@gmail.com

About This Document

This document describes the design of a calendar poll system like doodle, using the Java Messaging Service API of the Joram middleware. In particular, we use the Joram's API for point-to-point communications (through message queues) to implement the poll system.

Design Complexity

Consider the system activities from the point of view of what happens for any specific calendar poll from its creation to the time when a finalized meeting time is decided. From this perspective, participants in the system can be divided into three distinct categories (for any poll object):

1. A unique **initiator**. The initiator initiates the poll, records the responses of the responder participants, and makes the decision on the finalized meeting time.
2. A set of **responders**. They are responsible for responding to a poll by specifying a "yes", "no", or "maybe" decision to each proposed meeting time.
3. The remaining participants, who are unaware of the poll.

Of course, a participant plays different roles among the above, for different polls.

From the middleware service perspective, the complexity of the design arises from three sources.

- The system must handle the *communication* aspect of the polls. This entails effective use of Joram's point-to-point communication and message queues for (1) sending the initiated poll from its initiator to the responders, and (2) sending the response back from the responders to the initiator.

- Several concurrent polls may be active at any time, for instance with the same participant playing the role of initiator of one poll and responder to others. Thus there must be a way of keeping track, from the perspective of a participant and a poll, the role of the participant and possible actions.

- User communications at the user interface for a participant may be asynchronous with the communication among poll participants. Thus there must be a way for a participant process to handle user command when it is waiting for a response from a poll participant (and vice versa).

Design Decision

Our approach to handling the above complexities entails two key design decisions.

The first key design decision is to define a *poll object* for each poll that is created. This object is passed around to (and updated by) the different participants. The idea is that a poll object maintains all the relevant information for a specific poll at any time, including (1) a unique id, (2) the name of the initiator, (3) the list of participants, (4) how each participant has responded to the poll so far, (5) the status of the poll (whether it is still active or has been closed by the initiator), (6) the final decision on meeting time (in case of closed polls), (7) the participant sending the poll object (see below), etc.. Thus a poll initiator does not need to maintain any additional state in order to determine the status of a poll or any necessary potential action; all such information is embedded within the poll object.

Given the use of the above poll object, handling concurrent polls becomes easy. Every participant process maintains two lists: (1) an *initiator list* of poll objects for polls it has initiated, and (2) a *response list* of poll objects for polls for which it is the responder. No additional control state needs to be maintained at any process.

The only remaining design issue then is how the poll objects are passed around and how the two lists above are updated by the different participants. Our simple approach for communication is to maintain a *single* mailbox queue of poll objects for each process.

- The initiator creates a poll object, sets its status as "active", and sends it to the mailbox queue of the responders. (The set of responders is determined by the user interface.) The initiator also adds the object to its list of initiated polls.

- A participant p receiving a poll object o in its mailbox queue for which p is a participant but not an initiator, inserts it into the response list of poll objects for which p is the responder (possibly updating or removing any other poll object with the same id as o).

- A participant p periodically picks a poll object o from its response list and responds to it. This activity is controlled by the user interface of p. When this is done, the status of object o is updated to "responded" to record that p has already responded. A participant cannot respond to a poll twice; also, a participant cannot respond to a poll with status "closed" (see below).

- Suppose a participant p receives a poll object o in its mailbox queue from q. Suppose also that p is labeled as an initiator of o. Then p finds a poll object o' in its initiator list with the same id as o, and updates o' to incorporate the decision of q. Note that simply replacing o' with o is not sufficient since in the object o all the recorded decision other than that of q may not be accurate. Recall that a poll object includes information about the sender; that information is used to determine the process whose decision needs to be updated. Finally, the update is permitted only for a poll o' whose status is not "closed". Thus, update to an already closed poll is ignored.

- A participant p at any time can close the poll. This activity is governed by the user interface. This is done by updating the status of the poll object to "closed", and sending the updated poll object to each participant originally invited to the poll.

Finally, we briefly mention the issue of asynchrony between the communication among participants and the user interface. Note that since there is only one mailbox queue per participant, the concurrency involved is restricted to two: responding to the user interface and responding to a communication in the queue. We address this by using two dedicated threads for each process, one handling the user interface and another listening to the queue.

Implementation and Assumptions

We implement the system to faithfully conform to the above protocol. The above protocol is very general, with no assumption on the number of participants. However, the implementation makes a few usage assumptions. We explain the assumptions below.

- The system currently supports at most 7 participants out of the box. This restriction is imposed by the design decision to associate a dedicated queue for each participant. This is because in Joram the queue objects must be created a priori

in the Joram administration code and bound in Java Naming and Directory Interface (JNDI). For instance, in our implementation, the queue objects are created in the `ClassicAdmin.java` file distributed with Joram. The a priori creation of queues implies that one cannot change their number dynamically at run-time. In order to create more participants (and hence queues), the administration code must be appropriately modified.

- Participants must be uniquely identified. In our implementation, the participant names are provided in a configuration file (which associates the names with a unique id based on the position of the name in the configuration file). During run-time, the system prompts for the id of a poll client to be started. However, to our knowledge, there is no direct way in Joram to check that a poll client with the same id is not started more than once. We believe that this is a natural assumption to make in practice, where each participant may be provided a unique id in some other means (*e.g.*, through email address).

- If a participant does not vote on a poll, we make the default assumption that the corresponding vote is a "maybe". Finally, the poll initiator is permitted to pick any of the proposed meeting time as final, irrespective of the preferences of the participants.