Jay Almarode

Communicating between sessions

IN MULTIUSER ENVIRONMENTS, users often want to communicate with each other. In some cases, they want to broadcast their message to all interested parties; in other cases, they only want to carry on a dialog with one other user. In a system that supports transactions, an application sometimes wants to be notified when changes to particular objects have been committed. A client/server system already has the infrastructure to provide these services. The client and server already have predefined communication protocols, and the server has knowledge of all the clients currently logged in. This column discusses two kinds of client-to-client communications that can be supported by multiuser Smalltalk and shows how to use them to implement concurrent processing algorithms.

In a system that supports transactions, application sessions are committing changes to objects all the time. Many times an application needs to know when another concurrent session has committed a change to specific objects of interest. For example, a stock broker application may want to trigger some activity when the price of a particular stock has changed. Or, an inventory management application may need to initiate item purchases when the inventory dips below a specified threshold. A reservation system may want to be notified when a new reservable unit becomes available. In these cases, the application does not care who made the change; it just wants to be notified that a change occurred and which objects were modified.

In GemStone Smalltalk, class System provides protocol to receive notification when particular objects are modified. Each session that is logged in maintains its own 'notify set'. A session can register objects of interest by placing them in its notify set. This set only exists for the life of the session; that is, it is not a persistent object, but it is maintained across transaction boundaries. An application can add a single object to its notify set by executing System addToNotifySet: anObject or can add a collection of objects by sending addAllToNotifySet:. There is also protocol to access and remove objects from the notify set.

Once objects have been added to its notify set, there are two ways in which the session can receive notification. One way is to poll for the objects that have changed; the other is to install an exception handler. If the application installs an exception handler, it must first enable the ability to receive this error (it is not really an error, but the underlying implementation uses the error mechanism to interrupt execution). This error is enabled by sending System enableSignaledObjectsError. Whether polling or handling an exception, to find out which objects have been modified, the application sends System signaledObjects. This message returns a set of objects that have had changes committed to them and clears the signaled objects set for the next use.

The following section of code illustrates how to install an exception handler and get the changed objects:

```
"first enable the ability to be notified when
    changed objects are committed "
System enableSignaledObjectsError.

" now install an exception handler to catch the notification "
Exception
    category: GemStoneError
    number: (ErrorSymbols at: #rtErrSignalCommit)
    do: [ :ex :cat :num :args | | changedObjects |
        " get the objects that have had changes to them
          committed "
        changedObjects := System signaledObjects.
    ].
```

When adding objects to the notify set, an application must consider what part of an object will actually be written, so that the session will be notified when a change to that object is committed. For example, an RcCounter object (described in an earlier column) is actually implemented as a composite object, composed of multiple subobjects that each encapsulates a numerical value. It is the sum of all values in the subcomponents that actually make up the RcCounter's value. When an RcCounter is incremented or decremented, it is one of the subcomponents that is actually written. Consequently, to receive notification when a change to an RcCounter is committed, a session must place the root object and all of its subcomponents in the notify set.

Another kind of useful client-to-client communication

Using Smalltalk since 1986, Jay Almarode has built CASE tools, interfaces to relational databases, multi-user classes, and query subsystems. He is currently a Senior Software Engineer at GemStone Systems Inc., and can be reached at almarode@gemstone.com.

is when two sessions want to talk to one another directly and immediately. In GemStone Smalltalk, this is possible by sending a signal to another session currently logged in. A signal consists of a SmallInteger, whose meaning is agreed on by the participants in the dialog, and a sequence of bytes (a maximum of 1,023). As with the changed object notification mechanism just described, signaling is implemented using the underlying error mechanism. Consequently, a session must enable the reception of these signals by sending System enableSignaledGemStoneSessionError. A session can receive signals from multiple senders, and the signals are queued in the order received.

For a session to send a signal to another session, it must identify the other session by its unique session identifier, a SmallInteger. There are a couple of ways that a user can find out about other sessions currently logged into the system. To get the session IDs of all users currently logged in, you can send the message System currentSessions, which returns an array of SmallIntegers. For each session ID, you can send System descriptionOfSession: sessionId to get back an array of more detailed information. Among the pieces of information returned by this message is the name of the host machine on which that user is logged in, and the UserProfile object for that user. Getting information about other users is a privileged operation, so you must have 'session access' privilege to send these messages.

Once you have the identifier of the session to which you would like to send a signal, you can send System sendSignal: aSignalNumber to: aSessionId withMessage: aString. The receiver of the signal executes System signalFromGemStoneSession to receive an array of signal information. The array is empty if no signal has been sent. If a signal has been sent, the array consists of three elements: the session id of the sender, the signal number, and the bytes.

Because signaling uses the underlying error mechanism, a receiver can install an exception handler to be triggered when a signal is sent to it, or the receiver can poll for signals.

Signals are a simple mechanism that can be used to build complex behaviors that involve more than one concurrent session. One use of signals is to coordinate sessions for implementing concurrent algorithms. Implementing concurrent algorithms with individual sessions means that you are allocating tasks among multiple processes, each with its own dedicated Smalltalk interpreter and view of the object repository. Some care must be taken to make sure that each session's transaction point of view is reasonably up to date with the others. Usually this means that a session begins a new transaction as the first step in performing its part of the concurrent algorithm.

The remainder of this column describes a simple pair of classes that can be used for implementing concurrent processing. The implementation consists of one class, called WorkerBee, whose responsibility is to receive commands to do work, and another class, called QueenBee, which sends commands to multiple WorkerBee objects and accumulates the results of their work. A QueenBee uses signals to send commands to each WorkerBee and uses changed object notification to learn when each WorkerBee has committed its work.

The implementation of WorkerBee is fairly simple. Its main task is to execute a service loop, waiting for instructions from any QueenBee. Class WorkerBee defines a single instance variable, called amountToSleep that holds the number of seconds to delay each time through its service loop. This allows the responsiveness of each WorkerBee to be tuned. Note that the WorkerBee's OS process and resources used can be further controlled using configuration parameters as described in an earlier column on tuning. Each time, through its service loop, a WorkerBee checks if a signal was received. If so, it initiates some work based on the signal number. Because the meaning of signal numbers must be agreed upon by the sender and receiver, I use a pool dictionary shared by WorkerBee and QueenBee to provide symbolic names for different signal numbers.

The pool dictionary has entries with the following meanings:

#handshake	initiate an agreement to work for a QueenBee
#freeWorker	end the agreement to work for a QueenBee
#executeString	execute the given string for a QueenBee
#commit	commit the current transaction
#abort	abort the current transaction
#terminate	terminate the service loop of the WorkerBee

A WorkerBee must synchronize with a QueenBee before it does any work. In this simple example, a WorkerBee and a QueenBee perform a handshake in the following way: A QueenBee sends a signal initiating the handshake. Included in this initial signal is the QueenBee's name. This must be a name that the WorkerBee can resolve to get the QueenBee instance that sent the signal. If the WorkerBee is not already servicing another QueenBee, it returns a signal indicating its availability; otherwise it indicates it is busy. At this point, the WorkerBee is dedicated to a single QueenBee, waiting for commands. The implementation of the WorkerBee's serviceLoop is as follows (simple portions of this method have been omitted for brevity):

```
method: WorkerBee
serviceLoop

"Start up a loop, waiting for instructions."

| continue queen queenSessId
| continue := true.
" worker bee loop "
[ continue ] whileTrue: [ | signalArray |
    signalArray := System signalFromGemStoneSession.
    " if no signal was sent, sleep for awhile "
    signalArray isEmpty
        ifTrue: [ System sleep: self amountToSleep ]
        ifFalse: [
            | signalNumber signalSender signalBytes |
```

```
            signalSender := signalArray at: 1.
            signalNumber := signalArray at: 2.
            signalBytes := signalArray at: 3.

            " command to execute the given string "
            signalNumber = executeString
                ifTrue: [
                    " only accept commands from one queen "
                    (queen notNil and: [ signalSender =
                     queenSessId ])
                        ifTrue: [ queen addToHive: signalBytes
                        _execute ]
                ].
            " receive a request to work for a queen "
            signalNumber = handshake
                ifTrue: [
                    queenSessId isNil
                        ifTrue: [
                            queenSessId := signalSender.
                            " resolve the QueenBee's name to an
                              instance "
                            queen := System myUserProfile
                                        objectNamed: signalBytes.
                            System sendSignal: handshake
                                to: signalSender
                                withMessage: 'Available'
                        ]
                        ifFalse: [ " signal that the WorkerBee is
                                    unavailable "
                            System sendSignal: handshake
                                to: signalSender
                                withMessage: 'Unavailable'
                        ]
            ].

        continue := signalNumber ~= terminate.
    ]
].
```

The implementation of the QueenBee is also fairly simple. Class QueenBee defines three instance variables: its name, an array containing the session ID of each of its worker bees, and a bag in which each WorkerBee can place the result of its work. This last instance variable, cutely named hive, will be concurrently updated by multiple WorkerBees. To avoid concurrency conflicts, this variable contains an instance of RcBag. You may recall from an earlier column, an RcBag has concurrency semantics such that concurrent adders to the bag will not conflict.

In addition, the QueenBee wants to be notified when each WorkerBee has committed the result of its work to the RcBag. To accommodate this, the QueenBee places the RcBag and its subcomponents into its notify set. Once a QueenBee has issued the command for each of its workers to do some work, it can wait for notification of changes to the RcBag to gather results. The following code listing shows the methods to add the RcBag to the notify set, and to find all WorkerBees and perform the handshake with them:

```
method: QueenBee
addHiveToNotifySet

" put the RcBag and all of its subcomponents in the notify
  set "
System addToNotifySet: hive.
hive _doSessionBags: [
:addBag :removeBag |
    System addToNotifySet: addBag.
    System addToNotifySet: removeBag.
]


method: QueenBee
getWorkerBees

"Find possible worker bees, then perform a handshake to
 see if they are available for work. Set the array of worker
 bee's session id's with those that are available."

| possibleWorkers |
" find all users logged in as WorkerBee "
possibleWorkers
:= System currentSessions
    select: [ :sessId |
        ((System descriptionOfSession: sessId) at: 1)
        userId = 'WorkerBee'
    ].

" initiate the handshake "
possibleWorkers do: [ :sessId |
    System sendSignal: handshake
        to: sessId
        withMessage: name.
].

workers := Array new.
possibleWorkers size timesRepeat:
[ | signalArray |
    signalArray := System signalFromGemStoneSession.
    signalArray isEmpty
        ifFalse: [
            | signalSender signalNumber signalBytes |
            signalSender := signalArray at: 1.
            signalNumber := signalArray at: 2.
            signalBytes := signalArray at: 3.

            " check if WorkerBee was available "
            (signalNumber = handshake and:
            [ signalBytes = 'Available' ])
                ifTrue: [ workers add: signalSender ]
    ]
]
```

The mechanisms described in this column let one session find out about and communicate with other sessions. These mechanisms provide the infrastructure to build complex applications in a multiuser environment.