Developing Peer-to-Peer Applications on the Internet: the Distributed Editor, SimulEdit

Louis Thomas, Sean Suchter, and Adam Rifkin

CRPC-TR98737-S
January 1998

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Submitted January 1998; Published in Dr. Dobb's Journal 281, January 1998

Developing Peer-to-Peer Applications on the Internet:

the Distributed Editor, SimulEdit

Louis Thomas Sean Suchter Adam Rifkin

California Institute of Technology 256-80, Pasadena, CA 91125
{lthomas,triangle,adam}@cs.caltech.edu

Motivation

This paper describes a Java package that supports peer-to-peer communication (as opposed to client-server networking) on the Internet. We have used this package and its distributed data structure facilities to develop a distributed editing program.

More and more electronic documents are being created by groups of people. However, very few programs are designed to accommodate groups of people editing a document. Sometimes people sit around the computer screen and share the keyboard. The problem with this is that it is difficult for everyone to see the screen and for people to add ideas to the document as soon as they have them. Other times, each person modifies their own copy of the document. Unfortunately, these documents must be merged back together. The more changes that have been made, the harder they are to merge. We want to take a fresh approach. We want to allow many people edit the same file at the same time. Changes made by one person should be immediately visible to everyone.

Our original plan was a strict client server approach. This has advantages, including that the client would never have to see the whole file and that messages from client to server could just describe the requested change. In addition, simultaneous changes are ordered by their time of arrival at the server. Another idea we came up with was that the editor could be created as a true peer-to-peer application, where the document data belongs to the entire network of editors and not any of the individual editors. This means that if *any* editor were to suddenly drop from the network, the rest would be able to recover easily. From this, we developed the idea of a generic network data object. A network data object, or NDO, is an object that is shared among multiple processes communicating through a network and exists only as long as those processes exist. The data in an NDO may be manipulated or read by any process that is sharing the NDO. Changes made by one process are immediately visible to all other processes. The data disappears when all of the processes disappear, although a process may save a copy before exiting.

A text editor is an interesting distributed application to write because it has a variety of requirements. The cursors move quickly, and require quick synchronization across all the communicating processes. The lines of text are generally independent from each other, and each process is likely to only modify one or two at a time. The file as a whole will not change much. A user who is scrolling through a file without making changes would expect this to be a quick operation. How will the network package support all of these different requirements? Where will compromises be made?

General description of an NDO

A Network Data Object, or NDO, is a collection of data that is shared across processes connected by a network. From the standpoint of the processes involved, this is almost like having these variables stored in shared memory. In our implementation, all of the data that is shared is stored in types derived from a base class SharedVar. No SharedVar exists standalone; every SharedVar in an application is contained within an NDO. Using an NDO instead of more traditional networking methods, such as message passing, simplifies distributed application writing considerably. Instead of thinking of the program as communicating data between multiple computers, programming with an NDO is similar to writing a multithreaded, non-networked application. Any Java primitive type or class can be stored in a SharedVar, and the NDO will determine what messages need to be sent to keep all of the communicating processes up-to-date. In addition, because NDOs are lightweight and there is no need for interface compilers, skeletons, or stubs, there is less complexity for an application developer to master.

API Description

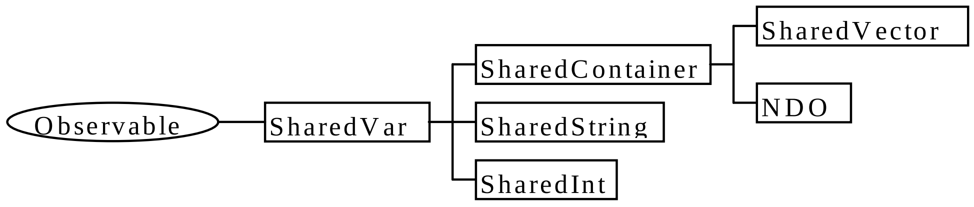

Figure 1 - Inheritance tree. Superclass of SharedVar is java.lang.Observable

As shown in figure 1, there are three key types involved in the NDO system: the classes NDO, SharedVar, and SharedContainer. Several other classes are also included in the package for both convenience and example code. In order to create objects for specific applications, the application programmer can add additional functionality by subclassing these base classes.

Class NDO

The NDO handles all global networking issues, as well as maintaining the list of processes that are sharing information. All SharedVar objects are contained within an NDO.

Class SharedVar

This is the base class of the information sharing system. Any piece of information that is shared across the network should be stored in a SharedVar. SharedVars also extend the java.util.Observable class, so that processes can track changes to the data through a callback. SharedVar is an abstract class. It must be extended in order for it to store data.

```
import java.io.DataOutputStream;
import java.io.DataInputStream;
import java.io.IOException;

public class SharedInt extends SharedVar {
  int _data;

  SharedInt() {
    super();
    _data = 0;
  }

  public SharedInt(SharedContainer parent) {
    super(parent);
    _data=0;
  }

public synchronized void readData(DataInputStream dis) throws
IOException {
    _data=dis.readInt();
  }

  public synchronized void writeData(DataOutputStream dos) throws
IOException {
    dos.writeInt(_data);
  }

  public synchronized void set(int data) {
    if (!locked()) throw new FieldNotLockedException("SharedInt not
locked.");
    if (_data != data) {
      _data=data;
      setModified();
    }
  }

  public synchronized int get() {
    return _data;
  }
}
```

Figure 2 - SharedInt source

Methods like set() or get() may be added to encapsulate modification events, and SharedVar’s abstract methods readData() and writeData() must be overridden so that the data can be transferred from process to process. The SharedInt class provides a simple example of how to make new shared data types. (Figure 2)

Class SharedContainer

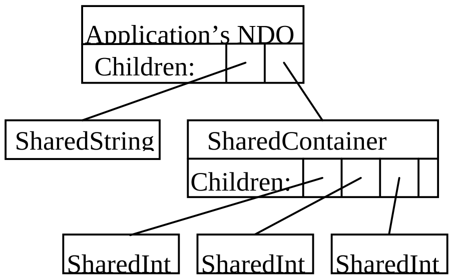

Figure 3 - Containment tree example

The SharedContainer is the class that allows for structured data, as illustrated in figure 2. A SharedContainer can contain any number of SharedVars, including other SharedContainers. A developer will probably extend SharedContainer so that the application can access the fields of a SharedContainer in a convenient way. For example, SharedVector is implemented using SharedContainer, and is more appropriate for common use because it allows access to the contained fields in an array-like fashion. SharedContainer uses a unique index for each new field that is added, and these indices are never reused. To store the references to the child SharedVars, the SharedContainer uses a java.util.Hashtable object to associate the integer indices with the SharedVars.

Initial setup using constructors

When an application program starts, it will create an instance of a subclass of the NDO object which has been extended to be appropriate for the application. One of the NDO's two constructors must be used. The first constructor, which takes no arguments, creates an NDO without contacting any other computers. The other constructor takes one argument: a location on the network. This constructor joins an already existing group of processes. During these constructor calls, the abstract method makeFields(), overridden in the subclass of the NDO, is called to create the basic fields that this NDO requires. Finally, after the NDO finishes setting up all of the fields and communicating with other processes as necessary, all fields are unlocked, the constructor completes, and the application can begin its main loop.

Working with fields

To modify a field, the application must have a lock on it. After obtaining the lock, the application can modify the data, and then it must call setModified() on the SharedVar. (In the example classes provided in the package, the call to setModified() is encapsulated in the set() method.) If the application wants to read the value of a field, it can do so whether or not the field is locked. If, however, the field is not locked, the value read is not guaranteed to be the latest one; once the field is locked, the application is guaranteed to have the latest copy of the data. Furthermore, an application is guaranteed that once it sees some version of a variable, it will never see an earlier version.

Locking fields

To write to a SharedVar or to ensure that it has the latest version of the data, the application must obtain a lock on the SharedVar. To do this, the application first calls the startLock() method of the NDO. Then the application calls the lock() method of the SharedVars it wants to modify or update. This registers that the application wants to lock that field but does not actually lock it. Finally, the application must call NDO.completeLock() to actually lock the fields. All of the inter-process communication is handled by completeLock() in a specific order to prevent deadlock, which is why completeLock() must be a separate operation. Only after the call to completeLock()

```
public class MyProgram {
  private MyNDO ndo;

  void doSomething() throws FieldDeletedException {
    ndo.startLock();
    ndo.field(0).lock();
    ndo.completeLock(); //This call will throw the
                        //FieldDeletedException if field 0
                        //of the NDO no longer exists

    ((SharedInt)ndo.field(0)).set(10);
    //SharedInt.set() calls setModified()

    ndo.field(0).unlock();

  }
}
```

Figure 4 - Locking example code fragment

returns can the application modify the SharedVar. As soon as the application no longer needs a lock, it should release it by calling unlock() on the SharedVar. This process of locking fields is illustrated in figure 3.

Adding and deleting fields

New fields can be added by obtaining a lock on a SharedContainer and constructing the new field with the SharedContainer as the value for the parent argument in the SharedVar constructor. The new field will be initially locked, so that its value can be set by the application.

A field can be deleted only when both the field and its container are locked. Once the locks are obtained, the removeField() method of the container can be called with the index of the field to be deleted as the argument.

Accessing fields

An important design consideration was to ensure that when fields were being added and removed from the containers, old fields not be confused with new fields. Imagine a scenario in which one process deletes the last field in a container, and then adds a new one. If another process only knows about the old version of the container (without the deletion and addition), it might request a lock on the old last field. Instead of giving a lock on the newly added field, an error is reported. This is done by assigning every field added to a container a unique index that is never reused. When an application requests a lock on a given field, it is guaranteed not to be confused with any other field for the entire existence of the NDO. SharedContainers are not arrays of SharedVars – that job is delegated to the SharedVector, which will reuse indices just like the Java built-in type java.util.Vector.

To access a given field of a SharedContainer, the application can call the field() method with the field's index as the argument. Alternatively, most application writers will want to store local references to the SharedVars they most commonly use in order to avoid lookup overhead.

How the NDO works

Every SharedVar is always owned by exactly one process. The owner of a SharedVar has the most recent copy of the data, and is responsible for giving the SharedVar to any process requesting a lock on it. Whenever a process obtains a lock on a SharedVar, it is then the owner. When a process unlocks a SharedVar, it remains the owner until some other process requests a lock on that SharedVar.

A process remembers to whom it gives the SharedVar. This is the process's best guess as to the current owner of the SharedVar. Each process has its own best guess as to who currently owns every SharedVar. When a process wants the lock on a SharedVar, it sends a request message to the process it thinks owns the SharedVar. If that process is not the owner, it forwards the request to whoever it thinks may own it. It is guaranteed that a request following this chain will eventually reach the true owner. When the message gets to the process that actually owns the SharedVar, several things can happen. If the process does not have the lock on the SharedVar, it immediately gives up ownership to the requesting process. If it does have the lock, it places the lock request in a queue. When the process unlocks the SharedVar, a response is made to the first request in the queue. The rest of the queue is transferred along with the SharedVar.

Individual ownership of every SharedVar makes most operations very simple, except for one: leave. When a process wishes to gracefully disconnect from the group of processes that are sharing the NDO it must ensure that it is not the owner of any SharedVars and that nobody thinks it is the owner of any SharedVars. When NDO.leave() is called, the process first obtains a lock on the NDO object. It then removes itself from the NDO's list of workers. If there are no other workers, nothing else must be done; the program can simply exit. Otherwise, a worker is selected from the list and this selected worker is given ownership of all of the SharedVars that the departing process owns. Then a message is sent out to every worker indicating that the departing process is leaving. Each worker must check to see if it has the departing process as its best guess for the ownership of any SharedVars. If a worker thinks the departing process owns a SharedVar, it sends a message asking who the departing process thinks owns the SharedVar. The departing process responds with who it thinks owns the SharedVar. When the worker is satisfied, it tells the departing process. When the departing process knows that all the other workers are satisfied, it transfers ownership of the NDO object to the worker to which it gave all of its other SharedVars. Finally, the departing process can exit, since no other process will ever need to communicate with it again.

Easy conversion to distributed applications

Using an NDO, most of the work of making an application distributed is identifying which variables are parts of the shared state and moving them all into a common location. Since the NDO package defines types similar to the basic Java types, conversion of individual data items is fairly simple. Of course, the conversion process cannot be done mindlessly! Two key issues must be understood and addressed by the application programmer: synchronization and granularity.

The synchronization problem is similar to the problem involved in writing a multi-threaded non-networked application. The NDO simplifies matters somewhat by reordering the lock requests of the fields by their indices so that deadlock conditions cannot occur. The application developer merely has to ensure that the application unlocks things once it is done with them. Since frequently passing a lock reduces performance, the data should be structured such that multiple processes are unlikely to need a lock on the same object at the same time.

The granularity problem is mainly a design issue. The application writer must remember that all data stored in a single SharedVar will be retransmitted every time it is changed. If it is common for only a small part of the data to be changed, it is appropriate to make a SharedContainer, which will contain individual SharedVars. Remember that SharedContainers need only be transmitted when *their* data changes, which primarily occurs when a field is added or removed. For example, in the SimulEdit application, every line is a SharedString, and all lines are contained within a SharedVector. We could have implemented it by making the entire buffer one SharedVar, but that would have required that each process get a lock on the entire buffer each time they want to modify anything. We could also have gone the other way and been very fine grained, having each line consist of a SharedVector of SharedChars. At the top level would be another SharedVector which would contain the individual lines. Not only would that have been wasteful of space, but it also would have been inefficient in transmission time, considering that editors rarely are used to *modify* characters. Rather, the user does things by deleting the old characters and then inserting new ones, which means that a lock on the entire line would be required anyway. For the editor application, the proper granularity was easy; other applications might require more thought.

Fault Tolerance

A fault-tolerant algorithm is being added to the base NDO package. Network and system faults - such as node failure - are handled transparently in an application that uses an NDO. Note that, unlike CORBA, our system does not depend on a server for transmission of data, so the failure of one process is not catastrophic.

Other applications

An NDO might also be appropriate for parallel calculation. A simple parallel computation engine could be written that would store tasks to be calculated in a shared list. Computers could then connect to this NDO, lock it, take a task out, unlock the NDO, and then think about that task for a while. When they have a result, they can lock the NDO again, and report the solution. This approach allows for computers to connect and disconnect from the computation group at will and select which tasks they want to perform. Best of all, since it is implemented with an NDO, it is completely peer-to-peer, with no server required.

The Editor Application: Overall description

In order to have an application to test the NDO classes and to use while thinking about how the NDO classes should act, we wrote a very simple text editor. It allows the user to edit a single file at a time, in a manner similar to the Windows Notepad. Cut and paste, as well as search and replace, are not implemented. However, SimulEdit does have a cursor that remembers what column it is supposed to be on, even if it is moved between lines of varying length.

The user can create a new file, read a file from disk, or join an already existing editing session. The user may save a copy of the current file to disk. Other people are not prevented from modifying the file while the user is saving it, so the user is not guaranteed to have the most up-to-date version of the file. The process can leave the NDO at any time if the user closes the file or quits the application.

The editor screen consists of the menu bar, the text grid (with its associated scroll bars) and the status bar. The status bar display messages to keep the user informed of what the editor is doing. It is also important because it displays the address of the editor, which other people need in order to join in the editing session.

The editor consists of about 9 classes: 2800 lines, 75K of source code, 50K compiled bytecode. Most of the difficulty was getting the Java Abstract Window Toolkit (AWT - The Java graphics library) to work and to work consistently on multiple platforms. We first developed the editor to work on a single machine, and then modified it to be distributed by using an NDO to store the data. It took only an hour or two to modify the editor to use the full NDO. When we ran a second editor and joined it to the first one, things worked almost exactly as expected the first time, with most of the bugs being in the NDO rather than in the editor or the interface between the editor and the NDO. This is a tribute to the simplicity of converting single user applications to distributed ones using NDOs.

How the NDO package was used

We chose to store each line of text in the editor as a SharedString, and to store all of the strings in one SharedVector. Next, we had to decide how we would represent the cursors. A cursor has an integer for its line and one for its column, plus other information like color. Since it was unlikely that the editor would ever lock a cursor’s line without locking its column or vise versa, we decided that all of the information relevant to one cursor should stored in a single SharedVar. We accordingly subclassed SharedVar to create a

SimulEdit Shared Cursor, or SEditSCursor. We then created an NDO that would represent an editable file (SEditNDO). It contains the list of lines, the list of cursors, and a SharedString for the file name. This is a reasonable way to lay out the data for any editor that wishes to keep each file as a separate object, *regardless of whether or not the editor is networkable*.

The NDO model makes the networking decisions very straightforward. We no longer have to worry about deadlock, and ensuring that the application has the latest copy of the data is as simple as locking the field. A price is paid, however, for not having to worry about deadlock. The application cannot lock one variable and then lock another without unlocking the first. This means that the application cannot lock one variable on the basis of the value of another. For example, when a user presses a key, that character should be inserted into the line the cursor is on. The application cannot know location of the cursor for sure until it locks the cursor. Once the cursor is locked, the application now knows which line to lock, but nothing can be locked without first releasing the lock on the cursor. Once the application releases the lock on the cursor, the cursor may move to another line. The application cannot assume that the line the cursor was originally on is still the correct one: perhaps another user inserted a line and all the line numbers have been shifted up by one. If the application naively inserts a character into the line where the cursor used to be, it would seem to the user as if the character was inserted into the line above the cursor! Our solution to this problem is to look at the current cursor position for a guess of which line to lock. Then, the application locks both the cursor and the line corresponding to this guess of the cursor position. If the cursor is not on the line that was locked, this process can be repeated with the cursor’s new position as the guess of the current line. Fortunately, the NDO handles data in such a way that this kind of locking and re-locking will usually be fast. Success is not guaranteed with this method, since

```
boolean bRetry;
do {
  bRetry=false;

  // Lock what we need
  int nLine=0;
  SharedString sszLine=null;
  while (true) {
    nLine=sscMyCursor.getLine();
    sszLine=m_seNDO.getLineObject(nLine);
    m_seNDO.startLock();
    scCursorList.lockAll();
    sszLine.lock();
    try {
      m_seNDO.completeLock();
      break;
    } catch (FieldDeletedException e) {
    // That's OK, try again.
    }
  }

  // make sure we get the line that the cursor is ACTUALLY
on.
  int nCursLine=sscMyCursor.getLine();

  if (nCursLine!=nLine) {
    bRetry=true;
  } else {
    // Do whatever you want to with the line and the
cursors
  }

  scCursorList.unlockAll();
  sszLine.unlock();
} while (true==bRetry);
```

every time the application releases the cursor to try again, another could insert a new line, moving the cursor. This is not likely to happen in a text editor. If this were a serious problem the application could lock all the lines. This is more expensive but guarantees success.

The greatest difficulty we experienced when converting the application to a distributed one is that in a single user editor, the data cannot be modified unless the user does it. When the editor becomes networked, however, other users may modify the data. Our application was not updating the screen properly when remote users made a change. We solved this problem by updating the screen on any change to the data, instead of just actions by the local user.

In the current implementation, some major improvements could be made in how the screen is updated. Currently, the NDO sends an update notification any time a SharedVar is changed. However there is currently no way of finding out what about it has changed. This is a bit of a problem in a SharedContainer, since we have no way of knowing what variables were added or deleted, or even how many. This causes the entire screen to redraw every time the editor gets a notification that a line may have been inserted or deleted.

Other design considerations

We spent lots of time wrangling with the AWT. If you want your graphical application to run across multiple platforms, you are going to have to test it on each platform. We wrote versions of the editor that ran fine under UNIX and hung under Win32, and vice versa.

We designed two AWT components that others may find useful, the TextGrid and StatusBar classes. The StatusBar displays its message in a nice box.

To display the text of the file being edited, we needed a grid of fixed-width characters just like a standard terminal screen. We also wanted to be able to control the foreground and background color of each character. The result is the TextGrid class. Creating a grid of fixed width characters presents a bit of a problem, since there is no reliable way to get a fixed width font in Java. Instead of asking Java to paint strings, we have to paint them one character at a time to enforce the fixed width. This is a very slow procedure. Using an off screen image was a clear solution because updates are flicker free and repainting the screen when the window is uncovered is almost instantaneous. If we only need to update a few characters, we can do it quickly. Scrolling the text that is already on the screen is very fast, although drawing an entire screen of new text is slow.

We encountered an interesting problem when trying to create an off screen image. Java.awt.Component.createImage() cannot be called until the component's peer has been created. This does not happen until the component is actually shown on the screen. Thus, any print requests that the TextGrid receives before it is displayed cannot be immediately placed in the off screen image and must be kept in some other manner. For this we used an array of character attributes which we immediately dispose of when the peer is finally added.

We also wrote a very short Assert class. We have no idea how we would have debugged the NDO classes without it. The relationships between the NDO object trees, their state variables, and the messages being processed are very complex and we found it impossible to remember all of them all the time. So, whenever we found a relationship that must hold true, we asserted it. Then, if we ever did anything later that violated that relationship, our code informed us.

Lessons learned

We learned a lot about writing distributed applications and many of the pitfalls of the java AWT. Not surprisingly, the largest cross-platform inconsistencies in the Java environment are the ones that interface with the hardware. We had difficulties with differences in implementation in both the java.net and java.awt packages. When one writes a Java program that is intended to run on more than just the development machine, careful testing must be performed on a variety of platforms and Java VM implementations.

Because of the peculiarities of focus on different platforms (especially Win32), we mange the focus in the Join dialog box manually. Instead of assuming that the AWT will give the focus to the appropriate component, the application saves a reference to the component that should have the focus in a member variable of the dialog named m_compCurrentFocus. To ensure that the proper component has the focus we just call "m_compCurrentFocus.requestFocus();". The application gives the focus to the first TextField as soon as the dialog is shown, so that the user can begin typing immediately. Any time the focus changes to a component that the application doesn’t believe is correct, such as the dialog frame or one of the panels used to organize the components, the application explicitly gives the focus to the correct component. When the user clicks in a component that can receive the focus, m_compCurrentFocus is set to point to that component. If, however, the user clicks on a component that cannot have the focus, such as the dialog background, the focus stays put.

Java components do not have any concept of a tab ring, the mechanism that Windows uses to allow presses of the tab key to move between dialog box fields. Since we are keeping track of focus, we can also catch tab and shift-tab key-presses and switch the focus to the next (or previous) component.

There is a bug in some Win32 implementations where clicking in a TextField does not produce the GOT_FOCUS event like it should. Therefore, anytime a key is pressed in a TextField, we also update m_compCurrentFocus.

Another 'feature' in some Win32 implementations is that a peer's handleEvent() is not called until after the parent's handleEvent(). Unfortunately, the Java specification does not say when a peer's handleEvent() is called. This makes it important for your handleEvent() methods to check not only the event's ID, but also the event's target. Since the TextField's peer has to get the KEY_PRESS event to actually add a letter, the KEY_PRESS event must travel not only thought the dialog's handleEvent() but also through the dialog's parent frame's handleEvent(). This was a problem when our editor frame was handling all key-presses (and just eating them while there was no open file). We had to modify the editor frame's handleEvent() so that it passed on KEY_PRESS events when the event's target was a TextField.

```
//-----------------------------------------------
// Clean up ASAP - inform parent that we've been dismissed
private void quit() {
    hide();
    dispose();
    getParent().postEvent(
        new Event(this, Event.ACTION_EVENT, null));
}
```

When presenting the user with a modal dialog, we want the main loop to wait for the user to dismiss the dialog, and then take further action based on what the user entered. We cannot block the event handling thread to wait for the dialog box, since doing so will lock up some implementations of the Java AWT. (An important exception is the FileDialog whose show() method does block until the dialog is dismissed.) One solution is to pass the dialog a reference to its parent and then have the dialog invoke a callback when it is dismissed. Another better solution, which we used in SimulEdit is to have the dialog post an event to the parent. This a good solution since the dialog doesn't have to know anything about the parent and the parent doesn't have to implement any special interfaces. All event processing remains in the event loop instead of scattered around in extra functions, and the parent gets a convenient reference to the dialog (in the target of the event) that it can use to extract the user’s input from the dialog. Since the Join dialog handles all events targeted at itself, the parent frame knows that any events it receives target at the dialog are messages from the dialog. When the dialog is dismissed, it posts an ACTION_EVENT to the parent with itself as the target. The parent frame can then act upon the user's response to the dialog.

```
// update as soon as possible
Graphics g=getGraphics();
if (null==g) {
  repaint();
} else {
  update(g);
}
```

The StatusBar often tells the user when the application is busy and will not be responding to more input for a while. Since repaint() requests are not acted upon immediately, the message may never appear if the VM is kept busy in another thread. What good is a "busy" message if it never appears? We would like the StatusBar to be repainted immediately. Calling update() on a component will make it immediately paint itself, which is what we want. To call update(), we must pass a Graphics object as the argument. We can usually get a component's Graphics by calling getGraphics(). Unfortunately, this call sometimes fails. If a component's peer does not exist, the call to getGraphics() will return null. In this case, there is nothing more we can do but wait, so we call repaint() instead.

```
//-------------------------------------------------------------------
// (override) When this is called, we can finally create the buffer.
public void addNotify() {
  super.addNotify();
  if (null==m_imOffscreen) {
    createOffScreenBuffer();
  }
}

//-------------------------------------------------------------------
// We can't create an image until peer has been created
private void createOffScreenBuffer() {
  Assert.isNull(m_imOffscreen);
  // Create the image
  m_imOffscreen=createImage(m_dCanv.width, m_dCanv.height);
  if (m_imOffscreen==null) {
    throw new RuntimeException(
      "TextGrid can't create offscreen image.");
  }
  // ... initialize, etc.
}
```

Another component that must worry about its peer is the TextGrid. To create an off-screen image, the peer must exist. Otherwise, the call to createImage() will return null. Until the off-screen image is created, any characters placed in the text grid must be stored some less convenient way. We want to know as soon as possible when we can create the off-screen image, which means we want to know when the peer is created. The solution is to override addNotify(). AddNotify() is called to add the peer to a Component. It is important that we call super.addNotify() so the peer actually does get created. Once the peer exists, we can successfully create the off-screen image.

When creating the SEditNDO, we wanted to create an NDO that held an entire file and all of its associated information. We also tried to encapsulate some of the NDO specific tasks into the class.

First, we created a class that extended NDO. This gave us all of the network functionality. Since the default constructor and the joining constructor had to do similar initialization, we separated this common code into another method, init(). In the joining constructor, the NDO requires a Place. Since the users think of Places as Strings, we decided that the SEditNDO constructor should take a String and convert it into a Place for use by the superclass NDO.

Next, we implemented makeFields(), the one abstract method in class NDO. We added the name, the cursor list, and the line list to the NDO, and added one line to the line list to create an empty file. Once this step has been completed, we have a usable (though not useful) NDO.

The init() method adds a new cursor to the cursor list. We do this for both constructors because every editor needs its own cursor. Overriding NDO.leave() is not usually necessary, but since we added a cursor when we started editing, we need to remove it when we stop editing. Overriding leave() ensures that our cursor will be removed at the right time. To create a cursor, first we lock the cursor list. Then we construct a new SEditSCursor with the cursor list as its parent, saving a reference to the new cursor for later use. Finally, we unlock the new cursor, since SharedVars are always created in a locked state, and we unlock the cursor list.

The SEditNDO has many accessor methods to make life easier for the editor. For example, the editor will often want to get the user's cursor. The user's cursor is one of many stored in a list. To make things faster, a reference to the user's cursor is saved so we don't have to look it up every time. The cursor() method of the SEditNDO returns the user's cursor without requiring the editor know anything about the list of cursors.

Finally, readFile() and writeFile() were written as part of SEditNDO to make loading and saving easy operations for the editor. ReadFile() locks all the lines and cursors in the SEditNDO. All of the cursors are repositioned to the start of the document, and all lines currently in the document are removed. Then lines are read on at a time from the InputStream passed into the readFile() method by the editor. For each line, a SharedString is constructed and added to the list of lines in the SEditNDO. Once all of the lines in the file have been read, all of the SharedVars (lines and cursors) are unlocked and all the other processes find out what happened. WriteFile() is simpler. Every line in the line list is written to an OutputStream passed in as an argument. Since it doesn't lock the lines, other users may make changes while the write is occurring. In an editor, however, this is not likely to happen, and the savings in overhead is large.

Acknowledgments

We used the info.net package for our message transport layer. The info.net package is part of the Infospheres Project, which does research on the systematic composition ("plugging together") of persistent active objects, which is being developed under Dr. Mani Chandy at the California Institute of Technology. This work was supported in part by NSF grant CCR-9527130.

Word count: 6137